OREGON GRADUATE INSTITUTE OF SCIENCE & TECHNOLOGY

Software Design for Reliability and Reuse: Phase I Final Scientific and Technical Report

Pacific Software Research Center
February 28, 1995

Pacific Software Research Center

Technical Report

Software Design for Reliability and Reuse: Phase I Final Scientific and Technical Report

Pacific Software Research Center
February 28, 1995

CONTRACT NO. F19628-93-C-0069
CDRL SEQUENCE NO. [CDRL 0002.11]

Prepared For:
USAF
Electronic Systems Center/AVK

Prepared By:
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
PO Box 91000
Portland, Oregon 97291-1000, USA

The Pacific Software Research Center is developing a new method to support reuse and introduce reliability into software. The method, called *Software Design for Reliability and Reuse (SDRR)* is based on design capture in domain-specific design languages and automatic program generation using a reusable suite of program transformation tools.

Phase I of the SDRR Project involved the creation of the reusable suite of transformation tools, and the use of these tools to develop a component generator for a particular domain. The domain selected for the project was Message Translation and Validation (MTV), a component of C^3I systems. The MTV generator (MTV-G) was developed from July 1993 to September 1994. MTV-G was subsequently used in a software engineering experiment comparing MTV-G with a state-of-the-art, templates-based development method. This experiment took place October 1994 to January 1995. Phase I of the SDRR project was sponsored by AF/AQK and monitored by AF/ESC.

This report documents the scientific and technical results of the Phase I SDRR project. The report comprises the following volumes.

Volume I **Software Design for Reliability and Reuse—Method Definition**
This describes the SDRR method for designing a software component generator.

Volume II **Measurement Final Report**
This describes the measurements used in the development of the SDRR tool suite and MTV generator.

Volume III **Design of the SDRR Pipeline**
This describes the design goals, structure of the pipeline, and design issues raised in the development of the SDRR pipeline.

Volume IV **Tool Survey**
This describes the tools required to support specific aspects of the SDRR method as it was applied in the MTV domain.

Volume V **Algebraic Design Language (Definition)**
This defines a new, high-level programming language called ADL, used as an intermediate language in the SDRR pipeline.

Volume VI **Results of the SDRR Validation Experiment**
This describes the quantitative results of the SDRR validation experiment.

Volume VII **Baseline Performance Measurements of Un-optimized Generated Code**
This describes the performance measurements conducted on the Ada code generated by MTV-G in the SDRR validation experiment.

Volume VIII **Technology Transition Plan**
This describes the plans for transitioning SDRR technology to government and industrial users. This volume is an update of the original SDRR Project Technology Transition Plan. Section 4 is the only updated section.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
2-28-95

3. REPORT TYPE AND DATES COVERED
Final Jun 93 - Jan 95

4. TITLE AND SUBTITLE
Software Design for Reliability and Reuse
Proof of Concept Project

5. FUNDING NUMBERS
C
F19628-93-C-0069

6. AUTHOR(S)
Dr. Richard Kieburtz - Center Director
Dr. James Hook - Project Director

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Pacific Software Research Center
Oregon Graduate Institute
P.O. Box 91000
Portland, Oregon 97291-1000

8. PERFORMING ORGANIZATION REPORT NUMBER
CDRL2.11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Materiel Command
HQ ESC/ENS
Bldg. 1704, 5 Eglin St.
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES Project Contributors

Dr. Tim Sheard	Lisa Walton	Dr. Francoise Bellegarde	Alex Kotov
Dr. Dino Oliva	Jeff Lewis	Dr. Charles Consel	Laura McKinney
Jef Bell	Tong Zhou	Walter Ellis	Linan Tong

12a. DISTRIBUTION/AVAILABILITY STATEMENT
A - unlimited distribution

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Pacific Software Research Center was sponsored by the Air Materiel Command to develop a new method to support reuse and introduce reliability into software. The method is based on design capture in domain-specific design languages and automatic program generation using a reusable suite of program transformation tools. The transformation tools, and a domain-specific component generator incorporating them, were implemented as part of a proof-of-concept project at the Oregon Graduate Institute of Science and Technology. A software engineering experiment was performed on the generator to assess its usability and flexibility.

This final report documents the Software Design for Reliability and Reuse (SDRR) Method, the Algebraic Design Language, system design, a survey of tools used and developed, baseline performance measures on instances produced by the generator, analysis of experiment results, final measurement report, and an SDRR implementation plan.

14. SUBJECT TERMS
Software Reliability Reuse <Program Generators>
<Formal Methods> <Program Transformation> Experiment

15. NUMBER OF PAGES
239

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT

Volume I

Software Design for Reliability and Reuse—Method Definition

Software Design for Reliability and Reuse
Method Definition

R. B. Kieburtz

February 27, 1995

This document describes the SDRR method for designing a software component generator. It gives the view seen by a team of computer scientists responsible for producing a design for a new class of applications. The view of a domain expert who will use the generator to create a specific software component is far simpler. A generated software component will typically be incorporated in an instance of a domain-specific architecture.

1 The SDRR design team

An SDRR design team includes a number of specialists. The specialists need to coordinate their efforts but are able to work independently and concurrently on their assigned tasks. Although the roles are described as those of individuals, a single person might play several roles on a small project, or a single role might be shared by several persons on a large one. The members of the team are:

- The design manager, who is a domain expert and has overall responsibility for the software design, its validation and documentation.
- A domain-specific language designer, who should have some knowledge of the applications domain but who primarily knows the principles of language design.
- The semantics expert, who understands formal semantics of programming languages and is expert at programming in the algebraic design language, ADL.

- The verification expert, trained in mathematical logic, who uses formal proof techniques to verify critical properties of a design.
- The implementation designer, who is expert in use of the implementation language and in choosing efficient data representations. This person also needs to understand the software environment in which a generated software component will operate.
- The transformations expert, who understands strategies for algorithm improvement.

2 The SDRR design concept

SDRR is a method for the design of flexible and powerful software component generators. With a component generator, the design encapsulated in the generator is the reusable artifact. Software components themselves are not the basis for design reuse. When a subsequent application or version of a design is needed, design modifications are made to the specification that is input to the generator, and a new software component is generated automatically. This allows each design change to be made at a level of abstraction at which details of the software irrelevant to the change are not seen. It also allows the applications designer and the software maintainer to use a design language that is expressive of the domain of application, rather than to encode the design in a wide-spectrum programming language.

Component generators achieve their greatest advantage for the design of families of well-understood software modules that are needed in many particular instances. If a software component is anticipated to be a one-off instance, dissimilar to any existing design, to be used once and never modified, then developing a component generator to produce it has little advantage. However, this is almost never the case. Most software modules have family resemblances to other, related modules and will undergo use and modification over an extended life cycle that requires their design to be maintained and updated. SDRR is intended to produce software components that can evolve reliably and inexpensively over an extended life cycle.

2.1 Steps in the design of a software component generator

The design of a component generator by SDRR proceeds in a series of steps that are carried out by the specialists of the design team, in cooperation with one another. Steps 1–3 must be performed in sequence.

1. Perform domain analysis to determine the requirements of the intended application. This step is common to all software development; it is not unique to SDRR. It needs the attention of a domain expert.

2. Formulate a domain-specific design language (DSDL) in which to express the parameters, operations and constraints necessary to meet the requirements of the domain application. This is a task for the domain-specific language designer, with the collaboration of a domain expert.

3. Formalize a computational semantics of the DSDL in terms of ADL, the algebraic design language used in SDRR. This task requires the semantics expert, a computer scientist with advanced training in formal aspects of programming languages and software design.

After these steps have been completed, the following steps may be performed concurrently.

- Prove critical properties of the formal semantics. This is the task of the verification expert. This task does not imply the need for a comprehensive "proof of correctness" of every aspect of the DSDL semantics, but it does offer an opportunity for formal verification of those properties of a design that are deemed to be of critical importance. Some properties can be verified independently of one another, or may be verified incrementally. Verification may be considered to be an off-line task, in that progress towards building an implementation does not have to await its completion.

- Design an implementation. SDRR implementations are stereotyped. An implementation expert designs a set of implementation primitives that specify how the computational semantics of the DSDL are to be realized in terms of a target programming language. For DoD applications, the target programming language will ordinarily be Ada. An implementation design is specified through a set of *implementation templates*, which are usually retrieved from a library, rather than designed for a specific application. An implementation design is tailored to an application by an *environment specification* that is also provided by the implementation designer. The environment specification details the interfaces to target language libraries and to other software or hardware modules present in a system architecture.

- Formulate tactics for performance improvement by program transformation. This is the task of the transformation expert. Performance improvement is obtained through the use of automated program transformations that are applied during the course of program generation. These transformations are mathematically based and are guaranteed to preserve the computational meaning of the ADL-specified semantics. The transformation expert designs a control scheme, or tactic, for application of these transformations to ensure their effectiveness. When an SDRR-designed program generator is applied to a DSDL specification, it automatically applies the necessary transformations by following the prescribed tactic.

3 Domain-Specific Design Languages — DSDL's

A domain-specific design language is used to formally specify software designs. It is a formal language that is expressive over the abstractions of an application domain. A DSDL may be wholly or partially declarative or it may be a functional language with libraries of functions specialized to the application domain. Common examples of DSDL's are:

- Tex and Latex, text formatting languages.
- Mathematica, an extensible language for mathematical modeling.
- Schema description and query languages for databases.
- Layout languages for prettyprinting the text of computer programs.
- A message format description language for the message domain of military C^3 systems.

An advantage of using a DSDL is that a domain expert can express domain-specific concepts directly, rather than encoding them. This allows the domain expert to formalize the specification of a software solution immediately instead of communicating a specification informally to a software specialist who may be less familiar with the intended application.

3.1 Designing a DSDL

A DSDL is defined by a computer scientist in consultation with a domain expert. In the design of a DSDL, a dialogue is necessary between the two in order to settle three important issues:

A. To clearly identify the principal conceptual abstractions of the domain. For example, in a language for formatting mathematics, the essential abstractions might include expressions, fractions, vectors, matrices, etc.

B. To formally define a language of terms to represent these abstract concepts. A term language can be defined in terms of a syntactic phylum [1] for each conceptual entity. The formal definition of a syntactic phylum is done through the use of a context-free grammar. Each abstract grammar generates a context-free language. These languages provide the means to express instances of domain concepts and of relations among them.

C. To interpret the relations among the principal conceptual entities. This interpretation is initially given by the domain expert in an informal manner, by describing the relations in natural language (English). The computational content of this description will later be elaborated by giving a formal semantics to the DSDL. For example, if the DSDL were a language for formatting mathematics, the relations among entities might consist of rules or constraints that govern the two-dimensional layout of their presentation within a rectangular window.

When designing a DSDL two important criteria should be kept in mind: (1) the DSDL must be intelligible to a domain expert, and (2) the formal semantics must allow a specification

[1] A syntactic phylum is a top-level category of grammatical forms, such as *subject*, *predicate*, *object* in the syntax of a natural language.

expressed in the DSDL to be translated into effective procedures that realize the specification. Typically, such a specification will provide static or dynamic constraints on an artifact of the application domain, or will specify its dynamic behavior.

Often, a graphical user interface (GUI) can be used to advantage to help an application designer to formulate an application design in the DSDL. With a well-designed GUI, the application designer does not need to "learn another language" in order to use the DSDL. The GUI takes the place of the "surface syntax" of the DSDL, providing instead the visual guidance of highlighting, windows and menus to guide the application designer to the desired structure of a formal specification. In the SDRR method, a GUI has very limited responsibility for checking data validity. It might, for instance, enforce a restriction on the number of characters in a fixed-length field of text but it would not be responsible for checking that words entered into such a field were valid or were spelled correctly.

3.1.1 Example—a message specification language

The message specification language (MSL) developed by PacSoft to specify the logical structure and data formats of messages in Air Force C^3I systems is an example of a DSDL. In this language, the structure of a message class is described by first giving the logical types of the fields that can appear in a message, followed by a record type that indicates how the fields are assembled to form an entire message. The translation of a message from an external representation as a sequence of bytes to its logical representation is given by declaring a translation action (called an EXRaction in MSL) for each field and for an entire message.

The example in Figure 1 illustrates the use of the MSL language. The entire description of message structure and translation actions is declarative rather than procedural. A type specification enclosed by square brackets designates a set of named alternatives. A type specification enclosed by curly brackets designates a record comprised of named fields. A translation action expression such as `Asc 3 | "ESC"` prescribes that a primitive reader function should read three Ascii characters from the input sequence, and that these characters are expected to match the string "ESC". If the match fails, the next listed alternative translation will be attempted.

```
(* Example message specification in MSL
 * a message contains three fields:
 * Name     - A variable-length field terminated with a space
 * Location - A four character fixed length string with the
 *            following possible values: "OGI ","ESC ","STSC"
 * ZipCode  - A five byte integer
 *
 * Logical types for the fields in a message: *
 * ----------------------------------------- *)
type NameType = string;

type LocationType = [Electronic_Systems_Center,
                     Oregon_Graduate_Institute,
                     Software_Technology_Support_Center];

type ZipCodeType = integer(00000..99999);

(* Logical type for the entire message *)
(* ---------------------------------- *)
message_type MType = {
        Name     : NameType,
        Location : LocationType,
        ZipCode  : ZipCodeType};

(* EXR -> LOG translation functions  *)
(* -------------------------------- *)
EXRaction to_Name : NameType = VAsc ".";

EXRaction to_Location : LocationType = [
        Electronic_Systems_Center          : Asc 3 | "ESC",
        Oregon_Graduate_Institute          : Asc 3 | "OGI",
        Software_Technology_Support_Center : Asc 4 | "STSC"];

EXRaction to_ZipCode : ZipCodeType = Asc2Int 5;

EXRmessage_action to_MType: MType = {
        Name     : to_Name,
        Location : to_Location,
        ZipCode  : to_ZipCode};
```

Figure 1: An example specification in MSL

3.2 Tool support for a DSDL

A DSDL has the syntactic structure of a context-free language. If it is given a surface syntax, then a parser generator tool such as *yacc* can be used to construct a translator from the surface syntax to its abstract syntax. If it is given a graphical interface, then a standard GUI design tool can be used. In either case, the structure of a design specification is determined by the abstract syntax of the DSDL.

4 Formalizing the semantics of a DSDL

The formal semantics of a DSDL is defined in terms of an algebraic design language (ADL). This semantics gives the DSDL a computational interpretation in which the relations between the principal concepts of a design abstraction are formalized.

The first step in the formal specification of a semantics is to specify a datatype that corresponds to the abstract syntax of the DSDL. To each operator of the abstract syntax there will correspond a data constructor of the datatype. The semantics of a term constructed with a given data constructor will be composed from the semantics of the subterms given as arguments to the data constructor.

A semantic specification will typically consist of two functions:

- a type checking function that starting from the abstract syntax, calculates a type expression for each term, and

- a translation function that, again starting from each term of the abstract syntax, calculates an expression in the semantic algebra representing the computational meaning of the term.

The type checking and translation functions are expressed in ADL and are evaluated on a declaration expressed in terms of the abstract syntax of the DSDL. They produce, respectively, a typing for the declaration and each of its subexpressions, and its translation into a semantic algebra. The semantic algebra is also expressed in terms of ADL, but this expression is not immediately evaluated. Rather, it is an abstract representation of the program that is ultimately to be generated. Thus the semantic specification actually constitutes a compiler from the DSDL into an ADL representation that also type checks the DSDL declaration given as input.

The control structure of an ADL program is specified through families of high level combinators. To each combinator there corresponds both a computation rule and a proof rule. The computation rules give an operational semantics to the ADL language and the proof rules give it a logical interpretation consistent with the computational one. Instances of the combinators are composed to form more complex function definitions in ADL. The laws obeyed by such functions are inferred by applying the proof rules of each constituent combinator.

Each operator of the abstract syntax of a DSDL is given a computational interpretation by a semantic function. The semantic function is well-typed in the type system of ADL, and is defined by cases on the data constructors of the ADL datatype derived from the abstract syntax of the DSDL. For each such case, the prescribed meaning of a DSDL fragment is specified by a computation programmed in ADL. The translation function from the abstract syntax of the DSDL to a semantic algebra is composed of the functions that interpret its fragments.

This programming technique uses the syntax of the DSDL to structure the specification of a computational solution. The resulting solution is compositional. Consequently, less attention is given initially to the efficiency of a solution than to the uniformity of its construction from its component parts. The goal is to specify a computation in such a way that it is amenable to formal reasoning, so that one can verify that it corresponds to the informally specified problem requirements. Algorithmic efficiency will be improved at a later stage by meaning-preserving program transformation of an ADL specification and by compilation into an efficient representation in an implementation language.

5 Verification of semantic properties

As noted previously, to every ADL combinator there corresponds a proof rule, which is either an induction or a coinduction rule depending on the nature of the combinator. The structure of the rule is dictated by the inductive (or coinductive) definition of the particular datatype whose homomorphism are defined by the given combinator. Since proofs are constructed out of these rules, it is possible to derive from a combinator and a proposition to be verified, the logical conditions that must be discharged to complete the proof. By automating the derivation of proof obligations, we obtain a goal-driven proof assistant for ADL.

Certain combinators require termination conditions to be proved. A termination proof requires the specification of a domain predicate and a well-founded ordering on the domain of the combinator. A domain predicate for a computable function f characterizes a subset of the values in the type to which f can be applied, for which the application of f can be proved to yield a result. A domain predicate for a function provides a verification condition for its applications, namely, that the actual argument given in each application satisfies the domain predicate. Termination proofs can often be given independently of other properties of a combinator.

The construction of proofs affords opportunity for human error, just as does the specification and design of programs. Verification by proof adds reliability not only because it involves formal reasoning, but because this reasoning can be checked by a mechanical proof assistant. A proof assistant for ADL is an important (but as yet unimplemented) adjunct to the set of design support tools.

6 Transformational Improvement

When the semantics of a DSDL is fully elaborated in ADL it is algorithmically effective. A component design specified in the DSDL can be executed as a rapidly constructed prototype. However, without further work, it is likely to have poor performance both in terms of execution time and space usage. Because the SDRR method encourages highly modular design of semantic functions in ADL, it produces a design that is easy to understand, to validate and to maintain. However, it engenders many more uses of function composition than might otherwise be necessary. Thus, control structures that might be shared are often duplicated, and intermediate data structures may be built and analyzed when they could have been avoided by careful programming.

To avoid paying performance penalties for modular design, SDRR employs extensive program transformation on the ADL specification. The transformations that are used are *meaning preserving*, which implies that they will never introduce errors that were not present in the original design. These transformations are, in fact, derived as instances of theorems in the algebra of ADL. There are known transformations that accomplish:

- deforestation—eliminates intermediate data structures;
- fusion—consolidates similar control structures;
- order reduction—replaces all uses of higher-order functions by equivalent, first-order functional representations;
- accumulator introduction—caches values to avoid recomputation;
- recursion elimination—replaces instances of recursion by iterative control structures.

Experience has shown that interactive direction of transformation steps is difficult to do effectively. Hence, transformations of an SDRR design are applied automatically. In applying the SDRR method, a human transformation expert supplies a tactic to control the automatic application of transformation steps. Transformations are directed by pattern-matching which triggers the invocation of embedded tactics.

7 Implementation Templates

An implementation is specified by a set of *implementation templates* and a given environment specification. An environment specification documents the system interface that will be seen by the generated software component. The interface may be constrained by type-correctness requirements. The functionality expected of the environment may be specified informally, or through a first-order logic or a software architecture description language.

The interface provided by the designed component includes the (typed) signature of its visible functions or procedures, together with the formal specification of the component as elaborated in the design.

Implementation templates are the forms for generated implementations. Rather than prescribing a fixed implementation stereotype, the SDRR method provides for parameterization of an implementation scheme via the use of a set of templates. Each template set provides macro-like translation forms for the operations of concrete algebras, such as booleans, character strings, and arithmetics. Concrete algebras also include the primitive access and construction functions of ADL datatypes. Implementation templates can be provided to generate code in different implementation languages, although for DoD projects, Ada will be the preferred language for implementation. A set of implementation templates must contain generic templates for free algebraic datatypes but it may also contain specialized templates for other concrete algebras that are declared as abstract data types. Through implementation templates, a designer can specify a hashed symbol table, for instance, as the implementation of an association list.

The source code representation of a set of implementation templates is typically quite small, on the order of a few hundred source lines, although a set of templates can grow if specialized implementations are specified for additional algebras that may be specified as abstract datatypes. Implementation templates are highly reusable, both because templates are copied many times during the translation of a single design from ADL to the target implementation language, and because a set of templates can be used in any number of specific applications.

8 Design inspections

SDRR design inspections are conducted by all the members of the group involved in a design activity. A inspection consists in critiquing different parts of the design to check for inconsistencies. The critical aspects to be inspected are the DSDL design, the semantics of the DSDL as given in ADL, the specified implementation templates, and confirmation that the environment specification has been satisfied.

Inspecting a DSDL is critical since its specification is initially given with an informal English description. Therefore, there is the possibility that it may contain ambiguities and misunderstandings. Allowing several people to study it often leads to a better definition of the language.

Prior to inspecting the semantics of the DSDL given in ADL, preliminary validation is obtained by type-checking the semantics specification. This provides early notification of all typing errors. For an informal validation of the semantics, the inspectors read and discuss the semantics. Formal verification is carried out by constructing proofs, using the proof rules that accompany each combinator. For reliability, proofs should be machine checked.

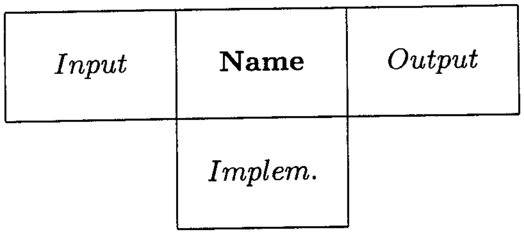
The design inspection committee also checks that the implementation templates meet all design constraints. Formal validation of the templates is done mechanically through exhaustive testing of each implementation function. A complete test procedure determines that each concrete algebra that may be used by a generator is correctly implemented in terms of the concrete algebras available in the target programming language. Notice that this does not imply a need to test the implementation of the target language—this is presumed to have been validated in advance of its use. After a set of implementation templates has been validated, it can be archived in a library of valid implementations.

Finally, it is also necessary to check that the environment specification has been met. This is a necessary step, even though integration testing will be performed, since exhaustive integration testing is always tedious and often impossible.

Design inspection of transformational improvements is only needed to determine the effectiveness of the transformation tactics. It is not necessary to review the correctness of the semantics transformations, since it is proved that none of the transformations change the meaning of the ADL forms, but affect only the performance of an implementation.

9 Tool support for SDRR

The preceding sections have defined the SDRR method independently of the software tools that support it. This section summarizes the design support tools that underlie SDRR. The tools can be envisioned in terms of 'T' diagrams, such as



in which

> **Name** is the name of the tool;
>
> *Input* is the language of its input;
>
> *Output* is the language of its output;
>
> *Implem.* is the language in which the tool is implemented.

When the output language of tool A matches the input language of tool B, these tools can communicate directly with one another. Otherwise, a translation of representation is necessary to compose two tools.

The languages indicated in this diagram are not all distinct. The following table summarizes the languages and their relation to one another.

Language	**Description**
DSDL	an acronym for a domain-specific design language
SML	Standard ML—a high-level implementation language
ADL	PacSoft's algebraic design language
CAML	a dialect of SML
CRML	an extension to SML
RML	a sublanguage of SML
RML_1	RML restricted to first-order functions

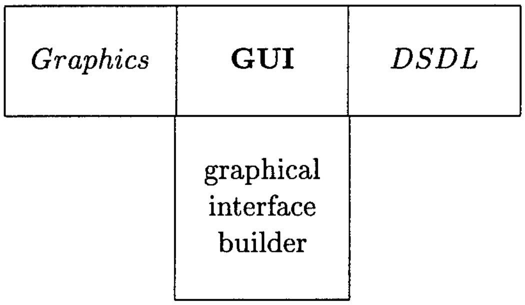
Of these languages, only SML and CAML have independent compilers supported by external organizations. The SML/NJ compiler is supported by ATT Bell Laboratories and by Princeton University. The CAML compiler is supported by INRIA, the French national laboratory for computer science. ADL is translated directly into SML and depends upon the SML/NJ compiler, including its type checking. CRML is an extension to SML, achieved through a set of library functions. It provides a structured interface to the meta-programming features of SML that make it convenient to manipulate programs as data objects, RML is an abstract syntax for a restricted sublanguage. It can be prettyprinted in SML syntax and compiled by the SML/NJ compiler.

The 'T' diagrams shown below for the tool *Astre* has been extended to show the interface modules that translate data representations. These interfaces allow these existing transformation tools to be seamlessly plugged into a composite sequence of SDRR transformation tools.

The tools are grouped into three sections, (1) those used to implement a DSDL, (2) those used to transform the ADL semantics of a DSDL into a simpler and more efficient program, and (3) those that translate the simplified ADL semantic representation into a program in the target implementation language. Tool group (2) will be standardized for the SDRR method and automatically invoked on each particular domain design.

9.1 Design capture in a DSDL

9.1.1 The graphical user interface



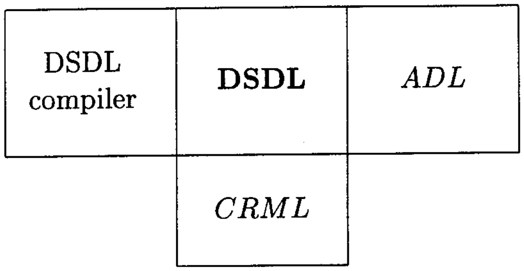
How provided: A GUI will be designed and constructed for each application domain.

Purpose: To provide an interface for the design of applications in the prescribed domain.

Capabilities:

1. Data entry—GUI embeds data into the phrase structure of the DSDL
2. Data editing—GUI supports editing of a specification.
3. GUI may perform some data checking and provides error recovery.

9.1.2 The DSDL Compiler



How provided: A DSDL compiler must be designed and implemented for each application domain.

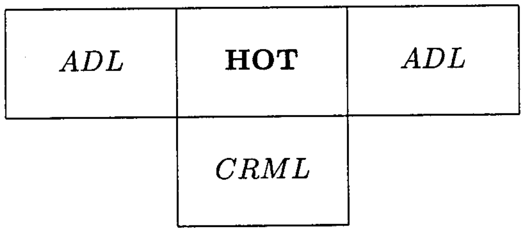
Purpose: To translate a domain-specific design specification into ADL.

Capabilities:

1. Sentences in a DSDL are translated into ADL.
2. Structural type checking of the DSDL input.
3. Concrete algebras can be specified as parameters of ADL modules.

9.2 Formal transformations for algorithm improvement

9.2.1 Higher Order Transformations



How provided: HOT is a reusable tool included in the SDRR tool suite.

Purpose: To improve algorithmic efficiency by applying the algebra of ADL to rewrite combinator expressions.

Capabilities:

1. To perform fusion for functions built with ADL's algebraic combinators.

9.2.2 ADL Translator

ADL	**Translator**	*RML*
	CRML	

How provided: The ADL translator is a reusable tool included in the SDRR tool suite.

Purpose: To transform ADL programs into (restricted) SML programs.

Capabilities:

1. Translates "core" ADL to SML.
2. Performs type reconstruction on an ADL program.

9.2.3 Order-reduction

Order-reduction consists of two parts that are applied in sequence. The first transformation, *Simplify*, accomplishes order-reduction in almost all cases and produces a direct representation of the transformed program. But there are cases on which the algorithm used in *Simplify* is known to be ineffective for fundamental reasons. When such cases are encountered, *Simplify* leaves higher-order functions in the program representation. The second transformation, *Firstify* is effective in all cases but transforms higher-order functions into data representations that may be translated into less efficient code. By applying the two algorithms in sequence, *Firstify* will affect only those rare cases on which *Simplify* is ineffective.

RML	**Simplify**	*RML*
	CRML	

How provided: **Simplify** is a reusable tool included in the SDRR tool suite.

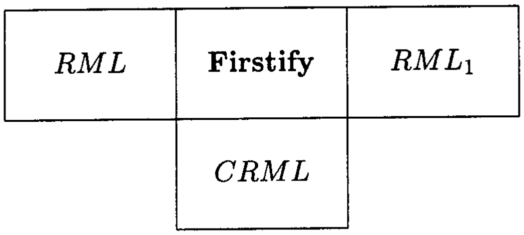
Purpose: To prepare RML expressions for further transformation and translation by:

- To ensure that all function applications are fully saturated (i.e. that there are no missing arguments).

- To specialize higher-order functions to first-order.
- To lift all declarations to top level.

Capabilities:

1. Provides dummy arguments to unsaturated function applications.
2. Specializes higher-order functions.
3. Lifts embedded function declarations to top level.



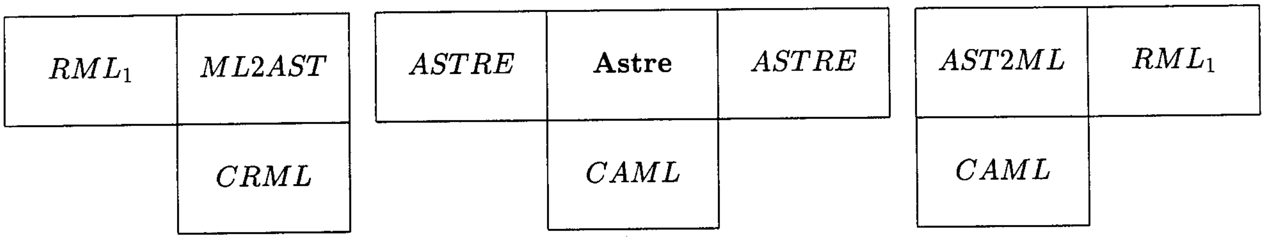
Purpose: To transform functional values into data structure representations so that the program can be transformed directly into an imperative language format.

How provided: **Firstify** is a reusable tool included in the SDRR tool suite.

Capabilities:

1. Processes expressions in RML, translating only functional value expressions.

9.2.4 Astre



How provided: **Astre** is a reusable tool included in the SDRR tool suite.

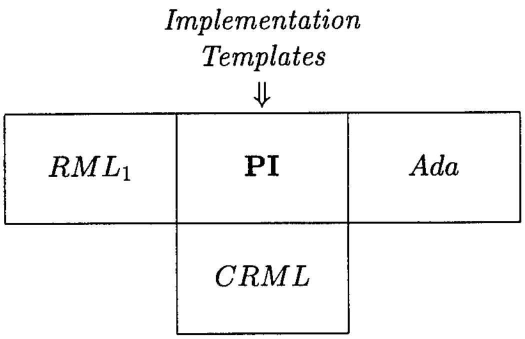
Purpose: To improve the computational performance of first-order functional programs.

Capabilities:

1. Replaces every function of multiple, individual arguments by an equivalent function of a single, tupled argument.
2. Provides automatic control of program transformations directed by programmed tactics.
3. Provides interactive control on demand.

9.3 Translation to imperative language for implementation

9.3.1 Program Instantiation



How provided: The Program Instantiator is a reusable tool included in the SDRR tool suite.

Purpose: To translate first-order RML programs into Ada.

Capabilities:

1. Generates single-assignment Ada functions for RML expressions.
2. Generates imperative Ada using state variables and exceptions.
3. Generates Ada packages.

A Algebraic Design Language

ADL is an abstract, high-level language with well defined mathematical properties. Its mathematical properties will support formal verification of properties in the design of the DSDL. Its high level of abstraction makes ADL suitable to explain the semantics of an arbitrary DSDL. ADL is sufficiently general to support specification of the semantics of a DSDL independent of an implementation. This allows for retargeting or reuse of software component designs with a variety of interfaces, and for optimizing the performance of implementations.

ADL is a very high level, typed functional language for designing software. In ADL, control is expressed through a family of type-parametric combinators. Certain combinators are parameterized with respect to datatypes, so that they can express the control associated with structural induction for any datatype. Additionally, ADL has coinductive types and there are coinductive combinators that express the control paradigms of iteration and search.

Control in ADL is completely specified through the use of its higher-order combinators, not through explicitly recursive function definitions or loops. ADL does not allow unstructured recursion, which may not terminate. A program whose termination properties cannot be verified does not have algebraic properties. Without algebraic properties, there are many program transformations and optimizations that cannot safely be performed.

Although control combinators can be expanded by being rewritten into recursion equations, these recursions are highly structured, and have special properties. One such property is that the recursion associated with an inductive combinator always terminates. The combinators admit inductive proof rules that provide structure for formal reasoning about properties of programs. The proof rules can be viewed as theorems about the algebra of the combinators. They also provide a basis for generic program transformation tactics.

ADL also has combinators that are not simply based upon primitive induction, but can realize more complex, transfinite induction schemes. When using these combinators, it is required to prove that the domain of each application satisfies a logical constraint ensuring termination of the computation. Only terminating computations are well-defined in ADL.

A.1 Programming with algebras

There are two main approaches to expressing software designs algebraically:

- **Abstract data types** (ADT's)

  ADT's specify the theory of a signature algebra as a system of equations. Typically, these equations refer to terms in particular datatypes and the algebraic theory is executable by a rewrite semantics. When an algebraic theory is expressed in terms of rewrite rules, it combines specification with the design of an implementation. The external view of an ADT hides both its implementation and its theory, revealing only its signature.

  The module system of ADL allows the importation of a concrete algebra without importing its abstract signature. Such an algebra is, in effect, an ADT. This mechanism is used, for instance to import an arithmetic algebra. The axioms of such an algebra extend the logic expressed by the proof rules of ADL.

- **Structure algebras**

  These are more abstract than ADT's, but they enjoy general properties useful for reasoning about programs. Structure algebras arise in the theory of universal algebras. The homomorphisms of these parameterized signature algebras are of particular interest.

  A structure algebra corresponds to a type constructor, parameterized with respect to a datatype. A great many program control structures can be characterized as homomorphisms of structure algebras. Among these are all the *reduce* functions for freely-generated algebraic datatypes, as well as more complex functions for non-initial algebras.

  A dual notion is that of a structure co-algebra. Many co-algebra homomorphisms correspond to the iterative control structures of conventional programming languages. However, their use in ADL is constrained by proof obligations that must be discharged to assure that iterations always terminate.

The control combinators of ADL are all based upon the homomorphisms of structure algebras and co-algebras. It is in this way that unbounded recursion is avoided. Instead of defining functions with recursion equations, operators are specified as structure-algebra homomorphisms. Hence, the algebraic properties of the programs are known immediately. These properties justify a variety of program transformations as consequences of the equational theory of an algebra.

A.2 Programming with monads

Two common techniques are used to compose software designs in ADL. The first is algebraic composition, which follows from the fact that algebraic programming is based upon the concept of multi-sorted signature algebras, parametric in a carrier set. When a signature algebra is instantiated with a particular carrier set that may also have algebraic structure, that structure is inherited and a composite algebra is formed. The other technique is semantic composition, using monads as the underlying structuring concept.

Monads provide a framework for structuring programming language semantics. Monads are algebraic structures that provide an abstract formalization of many programming concepts. It is through the introduction of monads that we are able to add more detail to the semantic domain. For example, state variables, I/O, exceptions, continuations, backtracking and concurrency can be added by interpreting a structure algebra in the appropriate monad. Furthermore, the desired semantic constructions can be incorporated incrementally into a design. The ability to incorporate state variables, exceptions and continuations into the semantics explicitly guides the final step of design; the translation from a purely functional, high-level design language to a lower-level implementation language with conventional, imperative features.

Monads have been advocated as a program structuring concept promoting reusability. By introducing monad definitions into ADL, we obtain a mechanism for generating composite combinators. We have successfully used monad composition as the basis for a new technique of design refinement.

A.3 Design by semantic refinement

Design refinement begins by specifying the names and types of semantic functions that realize the informally specified relations among conceptual entities described in a DSDL. As initially specified, these functions may not be effective. They may lack detailed, algorithmic definition. However, the control structure required for these functions can be specified in general terms, leading to the first refinement of high-level combinator definition. Refinement of the initial definition is provided by detailing state components and additional control refinements, such as exceptions.

Each control combinator is a higher-order function requiring a set of basic action functions as arguments. The type signature of the control combinator determines the types of the action functions. Each action function specifies the action of the combinator for a particular case or constraint of the data to which the combinator is applied. In this refinement approach, the action functions needed by a combinator can be defined independently of one another, as separable design tasks. Their designs may involve further steps of combinator specification and action-function refinement.

Selected action functions can be identified as policy parameters of the design. A policy parameter is a design parameter that specializes behavior to a particular application. A different policy parameter can be substituted to achieve a different specialization for a related application. Policy parameters are explicitly abstracted, creating derived combinators that incorporate committed design decisions but expose, through the policy parameters, design choices subject to change. In this way, the scope of variability of a design at the level of semantics is made manifest. This has important consequences for the maintenance of a software design.

Volume II
Measurement Final Report

Measurement Final Report
SDRR Proof of Concept Experiment

CONTRACT NO. F19628-93-C-0069

Oregon Graduate Institute
Pacific Software Research Center

Walter J. Ellis and Alexei Kotov

February 28, 1995

Executive Summary

In the Software Design for Reliability and Reuse (SDRR) project the Pacific Software Research Center (PacSoft) at the Oregon Graduate Institute of Science & Technology (OGI) explored a new approach to research in an academic environment. The goal of the project was to develop a new technology for efficiently building application generators; implement this technology in a real Air Force domain; and experimentally validate this technology by comparing it against other candidate technologies for improving productivity.

The PacSoft team successfully adapted military and industrial standard project management techniques not usually used in academia including detailed project planning, measurements, and quarterly customer program management reviews. The use of these techniques was a significant contribution towards the successful completion of the project and the product delivery on schedule and within budget. PacSoft used the recommended DoD core set of measurements (size, effort, cost, and schedule) augmented by additional measurements particularly suited for this project. Prior to this project, there was no measurement infrastructure at PacSoft. The measurement team developed this infrastructure to collect, store, and analyze using both commercial off-the-shelf products and "freeware" tools to collect, store, and analyze project data [7].

PacSoft used a new functional metric, the Capabilities metric, as a measure of product. This metric served as the basis for schedule tracking and estimates-to-complete. The early predictions for intermediate functionality were highly accurate. This allowed PacSoft to plan further project activities with confidence.

During planning, PacSoft developed a project Work Breakdown Structure (WBS) to separate the effort to develop tools from the effort to apply and verify the technology. The following major categories of effort were collected (Figure E-1):

1. Domain-independent application generator tools
2. Application of the tools to an Air Force domain, Message Translation and Validation (MTV)
3. Experimental validation of the technology
4. Measurement activities
5. Technology Transition
6. Management and Group Activities

PacSoft maintained control of quality by tracking defects discovered in the software during development and operation. Besides ensuring that open problems were visibly tracked to closure, PacSoft used the data to provide useful information on repair time (Figure D-2) and defect rates (Figure D-1). The average development defect rate per thousand lines of code over all products was 3.9. The product defect rate is 0.

The developed software had no failures in operation during the validation experiment. There were, however, performance problems encountered relating to the size of generated code.

Project change was monitored to collect information on change rate (Figure C-1), the impact of change (Figure C-2), and the distribution of changes in time (Figure C-3).

Average project productivity in development for the domain-independent application generator tools was 268 source lines of code per person-month. Average productivity for the tools specific to MTV-G application of the generator technology was 74 lines of code per person-month.

A few project members expressed concerns about the dangers of measurement. Some felt that management would use measurements like "Big Brother" to judge their performance. Others thought that data collection would consume too much of the project's limited resources. The acceptance of measurements gradually went from the suspicious--"Big-Brother" stage--to acceptance and fascination, as the information from early data became available. The project attitude today is "where else in the project and even in the institute can we apply this technology to improve our processes?" Measurements, a process now shared by all project members, contributed towards the success of the project by making visible all aspects of the project and product. With this vision, PacSoft management and technical personnel had solid information for early decision-making.

Contents

List of Figures

Introduction

We present a measurement perspective of the Software Design for Reliability and Reuse (SDRR) project at the Pacific Software Research Center (PacSoft) of the Oregon Graduate Institute of Science & Technology. This project explored both a new approach to software development and a new approach to research in an academic environment. The success of the SDRR project demonstrates that the use of project management techniques including measurements is important. With the use of these techniques the project was successfully completed and the product was delivered on schedule and within budget. PacSoft used the recommended DoD core set of measurements (size, effort, cost, schedule) augmented by additional measurements particularly suited for the task including predictability, flexibility, and usability.

In this document, we report the history of applying measurements to the project. At the same time we view the project processes and products through measurement. We highlight the important decisions that were made as a result of having measurement visibility. This information is valuable not only to future PacSoft efforts but to those organizations that may endeavor to start a similar measurement program.

We describe the project and the main results in Section 1. The SDRR project can be viewed as two experiments. The first is the experiment in process technology infusion, the meta-experiment, described in Section 2. The second is the experiment of application generation technology, results of which is described in Section 4. This section also contains the overview of management and group activities. The technical sections consist of two major parts. The first part is dedicated to the development of a reusable application generator. The second part is the application of this generator to one particular Air Force domain, Message Translation and Validation. The technology validation experiment demonstrated the benefits of SDRR by comparing it to a template-based technology, a previously-proven productivity enhancer. Some results of the validation experiment are presented in Section 4.7. More detailed information on the experiment planning and results can be found in the Experiment document [3]. The benchmarks of the performance of the generated code are presented in the Performance document [4].

The project-level application of measurements, including developing the infrastructure, is described in Section 3.

Section 5 contains a list of references. The Appendix contains a verification cross-reference index matching the final report accomplishments to the original plan document [9].

1 Project Description

1.1 Project Name, Description, and Location

The Software Design for Reliability and Reuse (SDRR) project was conducted in the Pacific Software Research Center at Oregon Graduate Institute of Science & Technology. The goals of the project were:

1. to develop a new technology for efficiently building application generators;
2. to implement this technology in a real Air Force domain;
3. to validate this technology experimentally by comparing it against other candidate technologies for improved productivity.

The time frame for the project was 7/1/93-1/31/95.

1.2 Project Personnel

The project technical personnel included

- 5 professors
- 1 postdoctoral researcher
- 2 staff members
- 5 graduate students
- 1 consultant.

All project personnel were working on the project activities part-time so the average level of effort for the project was 6 full-time equivalent persons (see Figure E-2 - total effort expended by week). This effort does not include the time spent by the professors mentoring graduate students that were developing part of the project software.

In the first half of the contract, the project was managed by a faculty member whose project responsibilities also included development of a tool and mentoring of a graduate student. This required the hiring of a professional staff member who could focus entirely on project management.

Over the 19-month project there was a significant change in project personnel. The existence of written plans, process definitions, and standards made the steep learning curve easier for the people joining the project.

1.3 Development Life Cycle

A three-part evolutionary prototyping life cycle was planned. Each tool was implemented in three phases, each phase resulting in a tool prototype. Prototype functionality was independently decided for each tool, but conformed to a basic structure. Prototype 1 tools included minimal functionality to support "manual" integration -- that is, interfaces between tools might require some manual intervention to be operable. Prototype 2 tools included minimal functionality to support full integration and the experiment. Prototype 3 tools included additional functionality, such as performance improvements and non-critical features.

1.4 Project Results

1.4.1 Investment

The effort expended to develop a reusable application generator was 46 person-months (7359 person-hours). This is a one-time investment. The effort to apply this technology to the Air Force domain was 12.6 person-months (2025 person-hours). We expect that application of the generator technology to a similar-sized domain will require a similar amount of effort.

1.4.2 Return on Investment

1.4.2.1 Technology Development

The software was developed and delivered on schedule and within budget.

1.4.2.2 Technology Application

The developed generator technology was successfully applied to a particular Air Force domain, Message Translation and Validation. We now know the complexity of such an application and we can predict the effort required to do similar applications.

1.4.2.3 Technology Validation

The technology validation experiment demonstrated 2.7 times productivity improvement using the SDRR technology compared to using templates. For details on the technology validation experiment refer to [3]. Brief analysis of the benefits of the use of this technology is presented in Section 5.1.

1.4.2.4 Technology Transition

PacSoft is in an early stage of transitioning the technology. The first stage of transition is maximum exposure. This was done using both traditional academic (papers, conferences) and non-academic (World-Wide Web, presentations to local industry) methods. The number of accesses to the PacSoft documents available on the World-Wide Web (Figures T-1, T-3) shows significant interest from different companies and research groups who could benefit from the technology. Figure T-2 shows technology transition activity by quarter.

1.4.3 Management, Measurement, and Process Improvement

The measurement program was initially started by customer request. Later, the benefits of the data collection became apparent. At this point, the group became the primary client of the measurement program. The use of metrics greatly increased visibility of the project processes. Analysis of the metrics data suggested possibilities for process improvement. Adoption of the measurement program by the team gave project management solid foundation for decision-making. Data collected over the course of the project allowed the team to make accurate predictions for the completion date for code development (Figure P-1) and software size (Figure P-2).

PacSoft realized that a measurement program should not be used in isolation. Instead, PacSoft developed a measurement program in a context of a continuous process improvement program. As the processes and the organization matured, PacSoft developed new objectives which required new and evolving measurements, sensitive to the needs of the team at all stages of software development, maintenance, and support.

2 The Meta-Experiment

Measurement in a research-oriented environment was as new as the technical aspects of the SDRR technology. Therefore the measurement activities had to be planned, monitored, and matured similarly to the activities aimed at the development of the reusable tool suite. Just as the SDRR technology led to validating experiments, measurement in this environment was itself a process experiment (which we called the meta-experiment) addressing the experimental questions:

1. Can a research team quantify the value of its new technology?
2. Can a research team quantify and compare the productivity and quality it achieves in developing the technology?
3. Can a research team use measurement to achieve technical and management goals?

These questions were stated in the Measurement Plan. This report presents the answers to these questions. Through the use of metrics, PacSoft was able to quantify the value of the new technology (see [3]). The project productivity and quality was successfully quantified (Sections 4.4 and 4.2). Measurements made the process more visible and provided a basis for early decisions. This added significant value to the project.

The challenge of the meta-experiment was the adoption by the team of the measurement and other project management techniques not usually found in an academic environment. The project results and this report demonstrate that both the experiment and the meta-experiment were a success. The product was delivered on schedule and within budget and measurements played a significant role in this success.

3 Measurement

3.1 Overview

Collecting the data was an ongoing project activity with built-in checks for validity. Analysis of the data was reviewed internally monthly for project feedback and presented externally quarterly as part of each Program Management Review (PMR). PacSoft delivered the measurement document in advance of the PMRs to provide adequate time for preparation. Additions or changes to reported measurement items were the subject of discussions at either PMRs or Technical Interchange Meetings.

The development process as represented by the WBS was instrumented by assigning task numbers to product life cycle elements. This produces effort matched to development activities of products. From this data PacSoft reported on the effort expended, progress in meeting schedule, and productivity, when the product was mature enough for size data to be available.

3.2 Project Experience with Measurements

3.2.1 Introduction of Measurement Program

The measurement program was initially started by customer request. At that time most of the project members had very little or no experience with measurements. So the introduction of the measurements raised several concerns in the group.

1. Project members were concerned that the collected data, especially the effort data, would be used to judge the performance of individuals by how much or how hard they work. It was very important that these concerns were addressed as soon as possible. The project addressed these concerns in two steps.

In the first step, project consultants explained to the group members the goals of effort data collection. The goals are much more important than finding out how many hours a project member worked on some particular day. The goal of effort data collection is to determine how much effort is needed to complete different tasks. This enables the team to accurately predict the effort required to achieve its future goals.

The second step was the correct use of data throughout the entire project. The project management committed to never use the collected data to evaluate performance of individuals. Each timesheet contained the following text: "*This information is part of the data for documenting the method and evaluating project management. It will not be used to evaluate the performance of individuals.*" The only project members who had access to the raw data were the project Administrative Assistant and the owner of the metrics process. Despite the fact that the project was small, the data was always presented in summary with individuals' names removed.

2. Another concern was that data collection would consume too much of the project's limited resources. Individual time reporting was new to the project. Adding recording of defects and changes was overwhelming.

The tools were selected to match the way the team worked and alleviated much of this concern.

The analysis tools were mature enough to make presentation of data a single-person task.

There were supporting and opposing forces to the use of measurement. Among the supporting forces were

1. the client;
2. the changing social contract for research;
3. previous experience with old methods that didn't work;
4. previous experience with ineffective teams of capable individuals;
5. recognition that this effort was larger than the previous center projects.

The opposing forces were:

1. radical change for an academic environments;
2. fear that planning would stifle creativity;
3. concerns about big overhead;
4. concerns about "Big Brother";
5. lack of experience.

PacSoft accomplished this transition one step at a time. Concerns were discussed at quarterly retreats and weekly meetings. The group identified risks and mitigation strategies for concerns about process and change. Measurements were used to evaluate the cost of the new activities.

3.2.2 Use of Measurement Data in the Process

The measurement program grew with the team maturity. The presentation of the measurement data to the group was slightly irregular at first. But then as the understanding of and the faith in measurements increased, the feedback became more and more important. A bulletin board dedicated to measurement was installed in the lab where weekly project meetings took place. The information was updated monthly at first. When the project entered its critical stage, the information was updated every week. This helped the group members to see the whole picture of the project and feel a part of the team. There is still potential growth in the use of measurements especially for the documentation and technical writing phases of the project.

3.2.3 Feedback

The flow of information between the project and the measurement process was two-way. The collected information and analysis was going from the measurement process to the group. The group, in turn, was communicating its needs back to the measurement process. As the team maturity and the faith in metrics grew, the demands for timely and versatile measurement information increased. This was perceived as a success of the measurement program in the group.

3.2.4 Resources Required for Measurement

The effort expended in measurement tasks was 6% of the total project effort (1229 person-hours). This figure agrees with industrial experience [1].

3.3 Infrastructure and Data Collection Details

3.3.1 Basic Infrastructure

The basic element of the measurement infrastructure was a measurable SDRR development process. This implicitly placed requirements on the project processes that they be well-defined and sufficiently delineated so that measurements are possible. This requirement translates at a lower level into defining measurable subprocesses. Subprocesses are defined by selecting a partition of the development process based on natural boundaries of the product development life cycles, which may differ depending on the maturity of the product.

Each subprocess has a well-defined initiator, which starts measurement simultaneously with the subprocess. Similarly each subprocess has a well-defined terminator, which brings a definite completion to the subprocess, allows definition of in-process measurements for that subprocess, and starts after-process measurements for that subprocess.

Examples of subprocesses, their initiators and terminators are in Table 1.

Subprocess	Initiators	Terminators
Prototype development	First, Design completion; Others, Completion of Previous prototype	integration test
Problem reports	reporter opens report	reporter verifies tests
Change Control	reporter opens change request	change tested and verified
Metrics data collection	infrastructure development	feedback of results

Table 1. Processes Initiators and Terminators

To implement measurements PacSoft instrumented both the process and the products and assigned ownership of the processes, products and related data to project members.

Process	Owner
Project Management	Laura McKinney
System Integration	Jef Bell
System Testing	Jef Bell
Change Control	Change Control Board
Measurement Process	Alex Kotov

Table 2. Process Owners

Product	Owner
MSL Compiler	Lisa Walton
ADL Compiler	Jeff Lewis
HOT	Tim Sheard
Firstify	Jef Bell
PEP	Jef Bell
ASTRE	Francoise Bellegarde
Chin	Tim Sheard
PI	Dino Oliva

Table 3. Product Owners

PacSoft designed forms and used automation whenever possible. PacSoft selected data repository and analysis tools including spreadsheets, databases, statistical analysis, annotated plot, and report generation tools. The purpose of the tools was to simplify data collection and analysis, facilitate feedback and reduce the load of data collection on the developers. In the beginning of the project, PacSoft paid special attention to create the measurement infrastructure.

The tool set used for measurement consisted of both commercial tools and "freeware" tools available at no charge [7]. When commercial or freeware tools were not available (e.g. code counter for Standard ML) or couldn't satisfy the project's requirements, custom tools were built.

The main collection and analysis tool for the metrics data was Microsoft Excel for Macintosh, version 4.0. This off-the-shelf tool was found to provide excellent data storage, analysis, and plotting capabilities.

3.3.2 Effort Data Collection

The effort data was collected using traditional timesheets that were turned in to the project Administrative Assistant once a week. The data was then entered into Microsoft Excel spreadsheet and transferred to the owner of the metrics process for integration and analysis.

Timesheet data for each developer consisted of the WBS number of the task the developer was working on, and the amount of effort expended (Figure E-3).

3.3.3 Defect Data Collection

Defect data was collected using a Free Software Foundation defect tracking tool called "GNATS". The tool was selected because of ease of use and adaptation. The centralized defect database was easily accessible from the standard *emacs* editor and e-mail environment that was used by most developers. The developers were even able to communicate through the notes section of problem reports. The system automatically notified responsible parties of problem reports arrival and changes in problem states. The tool also provided good extraction and selection capabilities which simplified the import of defect data to Microsoft Excel for summary and analysis.

When a system anomaly was discovered, an incident report was sent to the GNATS database. After analysis the incident reports (Figures D-7, D-8) were classified as unique problem reports, user errors or duplicates. Each problem report (PR) contained problem severity which could be critical, serious, or non-critical. "Critical" meant that this problem prevents the system from functioning normally and no workaround exists. These problems had to be resolved as soon as possible. "Serious" were the problems that would not prevent the system from functioning and problems for which workarounds were found. "Non-critical" problems were small and relatively insignificant problems such as typos or unclear diagnostic messages.

To report the current status of the project quality, we selected an aggregate parameter which was called "Problem Severity Factor" (PSF). This parameter was calculated using the following formula:

PSF = [number of critical non-resolved PRs] * 4 +
[number of serious non-resolved PRs] * 3 +
[number of non-critical non-resolved PRs]

This metric was selected because it allowed PacSoft to describe the quality status of the project in a slightly different way from the "total number of problem reports" metric. Problem Severity Factor emphasizes severity of the problem reports and increases very fast when the number of critical PRs increases. Figure D-4 presents both metrics together. The lower line shows total number of non-resolved PRs, the top line shows the Problem Severity Factor. The distance between the lines serves as an indicator of the severity of the non-resolved problem reports.

3.3.4 Change Data Collection

Change in the process was monitored in two different ways. Minor and trivial changes were simply communicated to the developer by the change initiator or a tester through the tracking system (see Section 3.3.3, "Defect Data Collection"). Proposed significant changes required approval and were assessed for impact through the project Change Control Board.

3.3.5 Software Size Data Collection

Software size in source lines of code (SLOC) was measured using a custom-made code counter (no other source code counter for Standard ML was found). The tool reported code size for a module in the number of executable lines of code, i.e. non-empty lines containing not only comments (ELOC), comment lines (lines containing only comments), and empty lines. The software size was measured on a weekly basis on a Friday night or during the weekend. Figure

S-2 presents the accumulation of the code during the project. This chart may serve as an indicator of project activity showing how different factors affect the code development.

One of the tools, ASTRE, was excluded from the history chart for several reasons. First, the tool was not included in the system delivered for the validation experiment. Second, at a certain point in the development, the ASTRE code was duplicated in a different location which made it very difficult to decide what part of the code should be used for size data collection.

3.3.6 Capabilities Data Collection

The capabilities metric was used to monitor the progress to the required functionality of the system. The capabilities metric and project experience with this metric is described in greater detail in [5]. Capabilities were defined as the units of functionality of software. Each tool in the pipeline had an associated list of capabilities that should be achieved. The capabilities were further categorized as critical or non-critical. The critical capabilities were defined as absolutely necessary for the proof-of-concept experiment while non-critical were additional functionality the absence of which would not jeopardize the feasibility of the validation experiment. Capabilities were the primary tool in monitoring the schedule and estimating time and effort to the project completion.

The choice of capabilities metric instead of other functional metrics such as function points was due to several reasons:

1. Application of the function points requires previous experience in software development in the same domain. Capabilities metric allowed us to use the data collected in the beginning of the project to make accurate predictions for the later project stages.
2. The function points metric provides a size/complexity estimate for the whole project or part of it. The Capabilities metric allowed a lower level of granularity, splitting functionality of each tool into measurable sections.

The progress to capabilities was monitored on a regular basis and reported back to the group and the monitors.

During the development of Prototype II, the list of capabilities and the schedule history allowed the project management to replan future development by removing some of the non-critical capabilities to meet the experiment schedule (see Figure C-2).

3.3.7 Data Collection -- Summary

The main risk of measurements was that gathering data might add too much overhead to design tasks. The risk was mitigated by automating the collection of measurements, and keeping the analysis simple on easily collectible data. With automation and human-factor considerations, the project avoided overburdening developers in data collection. Problem reports and minor change requests were blended in the normal e-mail environment the developers used. Measurement processes were written down to simplify adaptation of new project members to measurements. The following process definitions were created:

1. **time reporting** - described how to keep track of project time and how to submit timesheets;
2. **defect reporting** - described the steps to submit a problem report and the correct way to fill in its fields;

3. **defect tracking** - described the process of monitoring and tracking defects and state transition of a problem report.

Corresponding standard definitions were also created. These definitions included

1. **defect classification standard** - gave definitions of classes of problems such as faults, failures, and errors;
2. **time reporting standard** - defined the granularity at which the effort should be reported;
3. **defect reporting standard** - defined situations in which a problem report should be submitted, who should submit it and what should be described in the report;
4. **defect tracking standard** - defined who should change the state of the problem report, when it should be changed and who should close it when the problem is resolved.

Some of the processes and standards also had a list of Frequently Asked Questions (FAQ) attached to them.

3.3.8 Data Delivery

The data collected in the project will be scrubbed (i.e., names of individuals will be removed) and delivered to the National Software Data and Information Repository (NSDIR) simultaneously with the delivery of this report.

4 Project Results--WBS 1-6

4.1 Project Effort

The distribution of effort by task is presented in Figure E-1. The distribution of effort over time is in Figure E-2.

4.2 Project Quality

4.2.1 In-Process Quality

PacSoft maintained control of quality by tracking defects discovered in the software during development and operation. Defects discovered in testing or problems encountered by independent validation teams were ranked by importance, recorded in a database (see Section 3.3), and tracked through the problem life cycle. Although the defect reporting and tracking processes were working well, they may be improved further. Sometimes, there is a need for a method to reclassify defects or change their severity level. The process guidelines and standard definition for reclassification should be developed.

Besides ensuring open problems were visibly tracked to closure, PacSoft used the data to provide useful information on repair time (Figure D-2) and defect arrival rates (Figures D-1, D-9). The average defect rate per thousand of lines of code over all products was 3.9.

4.2.2 Inspections

PacSoft planned and conducted code inspections early in the project. The results from these inspections were recorded and analyzed. It was noticed that most of the defects discovered from these inspections were violations in the coding and documentation standards, not in defects in the operation of the code (Figure D-3). This was an indication that the investment in inspections was not working well for this particular project. In retrospect these code inspections would be considered informal walkthroughs by industrial standards.

After team members suggested ending these inspections, a heated discussion resulted in a compromise that at least two sets of knowledgeable eyes would review each product. These resulted in a new type of inspection called "shadow" or "buddy" inspections. These inspections were performed by a small group of people, usually two, that were familiar with the particular product being inspected. The results of these inspections were not recorded.

Late in the project when time and resources were tight, PacSoft revisited the use of formal inspections as the method to solve quality problems in the preparation of test data for the "Mock Experiment."

We present these incidents to show the growth in maturity of the project. And the use of measurement and sometimes the lack of measurement to guide us to process improvement. In this case we have learned the hard way the difference between informal walkthroughs and formal inspections. Also we have instituted a process of "shadow inspections," which has contributed to the success of the project. Without measurements, we can't report how much.

4.2.3 After-Process Quality

The developed software had no failures in operation during the validation experiment. There were, however, performance problems encountered relating to the size of generated code.

4.2.4 Quality Control -- Lessons Learned

Lesson 1. *Just do it.* Defect tracking is very important. It provides information on quality of the product, mean time to repair defects and allows prediction of the number of latent defects.

Lesson 2. *Be ready to customize.* No off-the-shelf defect tracking system is likely to suit all project needs. Therefore, both the defect tracking system and defect tracking scheme may have to be customized to suit the needs of each particular project and reflect the way each particular group works. In the beginning of the project PacSoft adopted a simple defect tracking scheme that was suggested by the GNATS system. In the process of using this system, the group found ways to customize the tool to better suit PacSoft's needs in defect tracking.

Lesson 3. *Be ready to change.* Once selected and adopted, the quality tracking scheme won't stay unchanged forever. The group's requirements to and expectation from the defect-tracking system changed, evolving with the group's maturity. We now feel that some aspects of the quality monitoring may be improved by customizing the classification of problems, and the number of states that problems pass through during the life cycle.

Lesson 4. *Find a way to communicate.* Quality metrics should be carefully selected to communicate the most important project information. At first, the Problem Severity Factor was used as the primary descriptor of the project quality. This is an important descriptor of the project quality, but sometimes this metric may be misleading because it doesn't contain an indication of its components (i.e. whether it represents a few critical problems or a lot of non-critical ones). A combination of PSF and the total number of non-resolved problems seems to be a better way to present project quality status.

Lesson 5. *Quality metrics may differ in different phases of a project.* For the maintenance phase, the customer priority level should be added to problem reports. A problem may have different priorities for the customer who is experiencing this problem, and the development/maintenance team.

4.3 Project Change

The two most significant changes in the project history were:

1. Replacement of the Schism tool with the Chin tool. The Schism tool existed before the start of the project and was perceived as mature and reliable. However, further technical investigation discovered that some aspects of this tool's design hampered its inclusion in the generator tool set. Furthermore, the author of the tool, who was initially an active participant of the project, left OGI, so the support for the tool could not have been easily obtained (technical aspects are described in detail in the document [6]. The decision was made to develop another tool internally to replace Schism. This change added 4 capabilities to the list of the system capabilities. The effort required to build the replacement tool was 430 person-hours. The impact in size was 4027 SLOC. Also, 332 SLOC were removed.
2. Prototype II replan. By May 1994 the metrics were indicating that the project was on track and the software development would be finished by the initial planned date (7/1/94) provided that the same rate of development could be maintained. However, the measurement history indicated that during the summer the project could expect a decrease in the effort expended on project tasks because of the summer vacations. This meant that the development rate would decrease and the initial plan may not have been met. The decision was made to replan the development by removing 8 capabilities not critical for the experiment (see Figure C-2, Replan Impact). Estimated impact of this change was: development effort: -1200 person-hours, schedule: -30 days, size: -2096 SLOC.

The following aspects of the project change were measured:

1. change rate (Figure C-1);
2. impact of change (Figure C-2);
3. distribution of changes in time (Figure C-3).

4.4 Project Productivity

4.4.1 Reusable Tool Suite

The average productivity for initial code development of the reusable tool suite was 951 source lines of code (SLOC) per person-month. The productivity rates for individual tools are presented in Figure S-3. High productivity for Firstify is due to the fact that the development of this tool was almost complete prior to the start of the measurements.

Overall productivity for the reusable tool suite development at the project scope was 268 source lines of code per person-month. Note: In the calculation of this parameter, the effort expended in WBS Task 2 (MTV-G) was omitted.

4.4.2 Domain-Specific Tools

The productivity for the MTV-G tool development at the project scope was 74 source lines of code (SLOC) per person-month. Note: In the calculation of this parameter, the effort expended in WBS Task 1 (SDRR) was omitted. Productivity number in this task is lower because it includes significant overhead relative to its small size, about 6000 SLOC. Also, the change rate for the MSL compiler was relatively high (Figure C-1).

4.5 Product Attributes

The attributes of products include size, effort, defects, change and schedule.

4.5.1 Reusable Tool Suite

The attributes of the reusable tool suite are in the following figures:

- The size of the reusable tool suite is in Figure S-1. The size is measured in Source Lines of Code (SLOC) of Standard ML (metalanguage) [2], the primary development language.
- Effort distribution on the project level is in Figure E-1. Effort expended on reusable tool suite development is detailed in Figure E-5.
- Average development defect rate per thousand of lines of source code for the reusable tool suite was 3.4. Defect rates and PSF of individual tools are in Figures D-5, D-10. There were no product defects discovered during 3 months of operation in the validation experiment.
- Average change rate per thousand of lines of source code was 0.3.
- Schedule of the tool development is in Figure P-3.

4.5.2 Domain-Specific Tools

The attributes of the domain-specific tools are in the following figures:

- The size of the domain-specific tools is 6000 SLOC of CRML (superset of Standard ML). The units of measure are detailed in the Design Document [6].
- The effort expended on MTV-G is 2025 person hours.
- Average development defect rate per thousand of lines of source code for MTV-G was 6.5. There were no product failures discovered during 3 months of operation in the validation experiment. The PSF of the MSL compiler is in Figure D-6. The MSL compiler defect rate is included in Figure D-11.
- Average change rate per thousand lines of source code was 5.9.
- Schedule of the tool development is in Figure P-3.

4.6 Management and Group Activities, Detailed WBS 6

4.6.1 Cost of Management Activities

The cost of management activities is detailed in Table 4. The management activities had certain overlap in effort which is shown in Figure 1. Detailed effort structure of WBS Task 6 (management and group activities) is in Figure E-4.

Activity	% of Total Project Effort Expended
Management	6.8%
Project Meetings	7%
Program Management Reviews	7%
Measurement	6%

Table 4. Project Management Cost

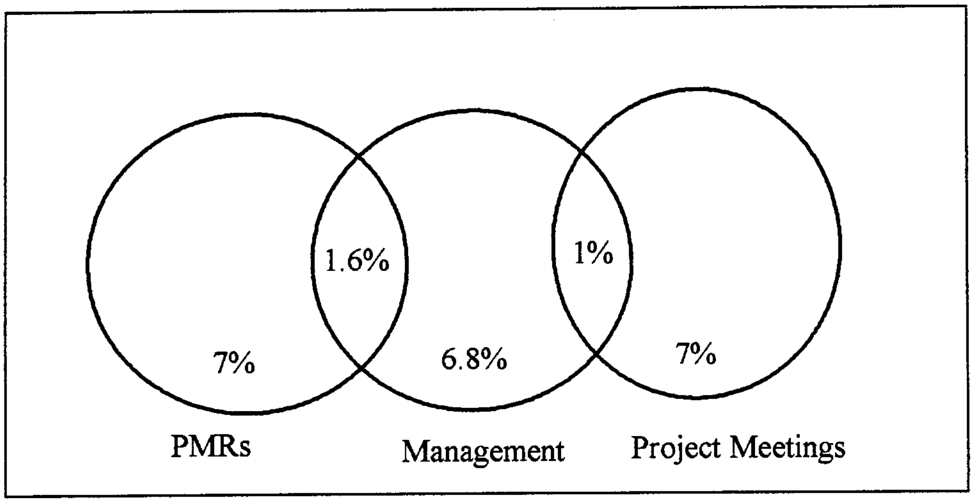


Figure 1. Management Activities Overlap In Effort

4.6.2 Group Activities: Weekly Meetings and Quarterly Retreats

Project meetings were held regularly each week. Total effort expended on project meetings and retreats was 1427 hours, or 7% of total project effort (with 1% overlap with project management, see Section 4.6.1). The team feels that benefits from regular project meetings were significant. The meetings allowed the project to stay on track, facilitated exchange of information between project members, and made individual progress of each project member more visible.

The activities performed on the project meetings included

1. discussion of weekly progress;
2. schedule planning and tracking;
3. task force planning;
4. Change Control Board discussions and decisions;
5. future activities planning;
6. PMR preparation.

The project meetings were augmented by off-site quarterly retreats. During the retreats, the group addressed long-term activities such as project planning, team building, and risk-assessment.

4.6.3 Quarterly Program Management Reviews

Quarterly Program Management Reviews (PMRs) contributed to better visibility of the project for the customer. In the PMRs and Technical Interchange Meetings (TIMs) customers received information about the project's progress towards goals. For PacSoft, PMRs and TIMs provided an opportunity to present technical work-in-progress. The central part of the PMR was management presentation on progress, status, risks and risk mitigations. PMRs would typically include presentations about the SDRR technology and status of the validation experiment. Also, the PMRs provided additional deadlines. In preparation for the PMRs project was baselined and assessed internally by the project members. During the PMRs, PacSoft was getting valuable advice and guidance from the customers.

Total effort expended in preparation and attending the PMRs was 1436 hours or 7% of total project effort (with 1.6% overlap with project management, see Section 4.6.1).

4.7 Validation Experiment

To demonstrate the benefits of SDRR, PacSoft ran a demonstration experiment with MTV-G to show the benefits of the technology in adaptability, flexibility, predictability, and usability. These experiments were comparisons of SDRR application generator technology to template technology. Matched experiments were run for SDRR and templates. The effort expended in preparation of the experiment was 9.7 person-months (1553 person-hours).
Detailed information on the experiment planning and results can be found in the Experiment document [3]. The benchmarks of the performance of the generated code are presented in the Performance document [4].

4.7.1 Results and Brief Analysis

Results of the experiment analysis show that the use of SDRR technology leads to 2.7 times improvement over templates technology in low-level design, coding, and testing stages of software development. These 3 stages comprise about 50% of a typical software development project. The 2.7 times improvement in 50% of the activities in a project would result in 32% increase in project productivity if all software in a project is generated using SDRR technology. Actual improvement will depend on the share of generated software in the total amount of software created in a project. Improvement may vary from 0% (when SDRR is not used) to 32% when all code is generated using SDRR (see Figure R-1).

5 References

[1] Tom DeMarco. Controlling Software Projects. Yourdon Press, 1982.
[2] Robin Milner, Mads Tofle, and Robert Harper. The Definition of Standard ML. MIT Press, Cambridge, Massachusetts, 1990
[3] Richard Kieburtz. Results of the SDRR Validation Experiment. In [8].
[4] Dino Oliva. Baselined Performance Measurements of Unoptimized Generated Code. In [8].
[5] Alex Kotov. Application of a New Functional Metric in a Research Environment. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.
[6] James Hook, Jeffrey Bell. Design of the SDRR Pipeline.
[7] Laura McKinney. Tool Survey. Software Design for Reliability and Reuse Project. Phase I. In [8].
[8] Pacific Software Research Center. SDRR project phase I final scientific and technical report, February 1995.

Appendix A

Verification Cross-Reference Index

General

Activity	Report Section	Figure
Use measurement for project control	3	All
Include measurements in the Program Management Review	3.1, 4.6.3	
Use DoD Core set of measurements (size, effort and software cost and schedule)	3.3.2, 3.3.5, 3.3.6, 4.1, 4.5.1, 4.5.2	S-1, S-2, E-1, E-2, E-3, E-4, C-2, C-4, P-1, P-2
Make comparisons between SDRR and other technologies	4.7, Experiment Report	
Measure steps toward the technology transition	1.4.2.4	T-1, T-2, T-3

Measurement Infrastructure

Activity	Report Section	Figure
Instrument both the process and the product	3.3.1	
Assign ownership of processes and products to project members	3.3.1	
Select data repository and analysis tools	3.3.1	
Build tools that do not exist	3.3.1	
Make clear the objectives of measurement to the members of the project	3.2.1	

Measurement, Analysis and Reporting

Activity	Report Section	Figure
Make data collection an ongoing activity	3.1	All figures
Present data internally monthly for project feedback	3.2.2, 3.2.3	
Review data externally quarterly as a part of each PMR	3.1, 4.6.3	
Present at PMR the DoD Core measurement of size, cost and effort at level 2 of the WBS	3.1, 4.6.3	

Report on effort expended, progress in meeting schedule and productivity.	4.6.3	E-1, E-2, E-3, E-4, P-1, P-2

Project Quality

Activity	Report Section	Figure
Record defects discovered in inspections	4.2.2	
Rank defects by importance, record in a DB and track through the problem life-cycle	3.3.3, 4.2.1	D-1 to D-9
Review the nature of frequently occuring problems for causal analysis	4.2.2	
Use analysis of faults to improve the development process	4.2.2	

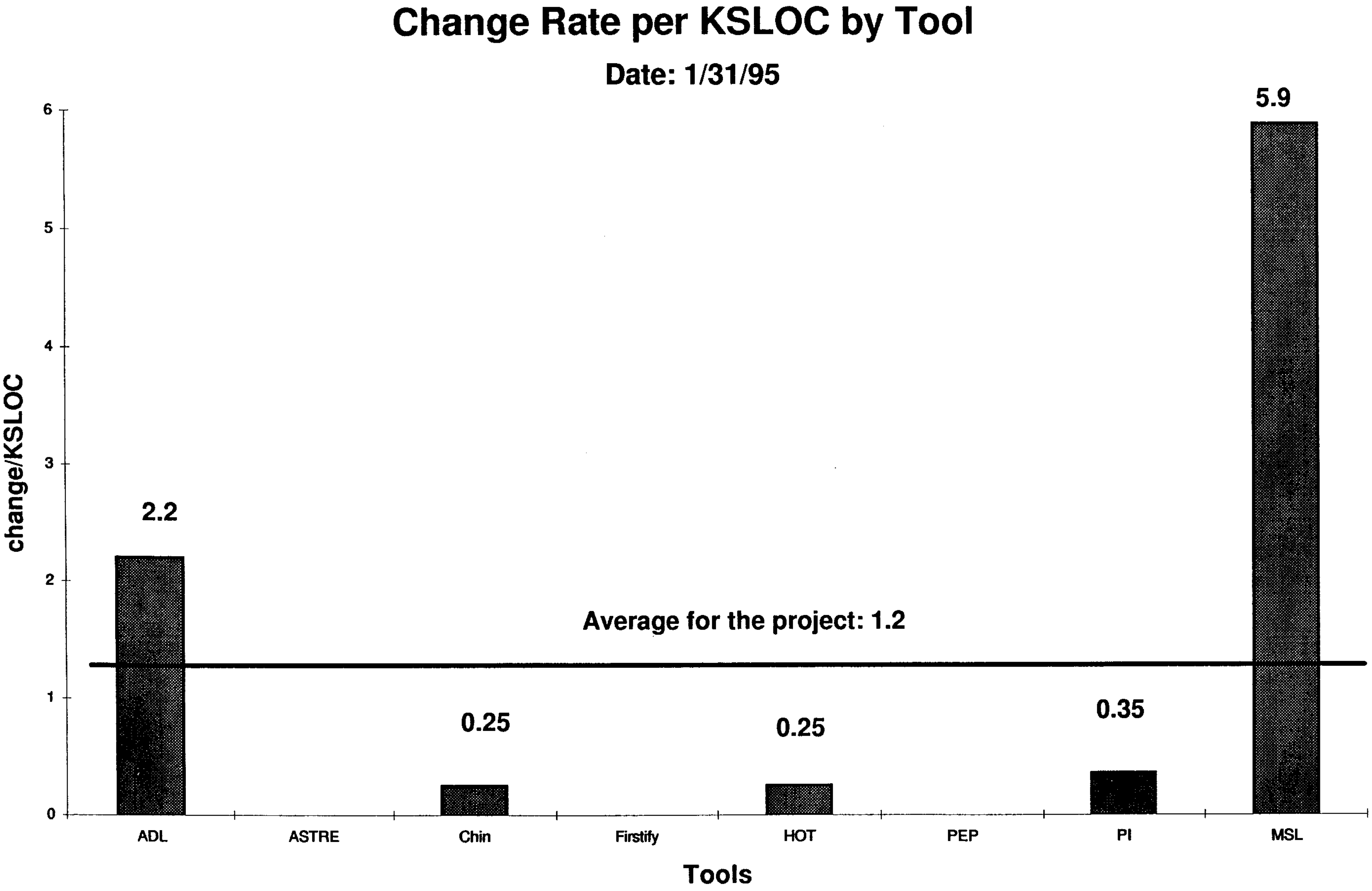
Change

Activity	Report Section	Figure
Record changes in a DB	4.3	
Monitor distribution of changes in time		C-3
Monitor rates of change		C-1
Monitor the status of change	4.3	
Monitor the impact of change		C-2

Process Tracking

Activity	Report Section	Figure
Summarize and report development of SDRR separately from tasks related to application of the method to MTV-G.	4.5.1, 4.5.2	S-1, D-5, D-6
Run demonstration experiments with MTV-G to show benefits	4.7, Experiment Report	
Track measurement tasks separately from development	3, 4.1	
Assess the success of technology transition both quantitatively and qualitatively	1.4.2.4	T-1, T-2, T-3
View the cost of management as a percentage of overall system effort and cost	4.6.1	E-1, E-4

Use these data to compare SDRR to other research and non-research efforts	3.2.4	
Scrub raw measurements to ensure privacy and make these data available at the end of the contract	3.3.8	



Change Rate per KSLOC by Tool

Date: 1/31/95

Figure C-1

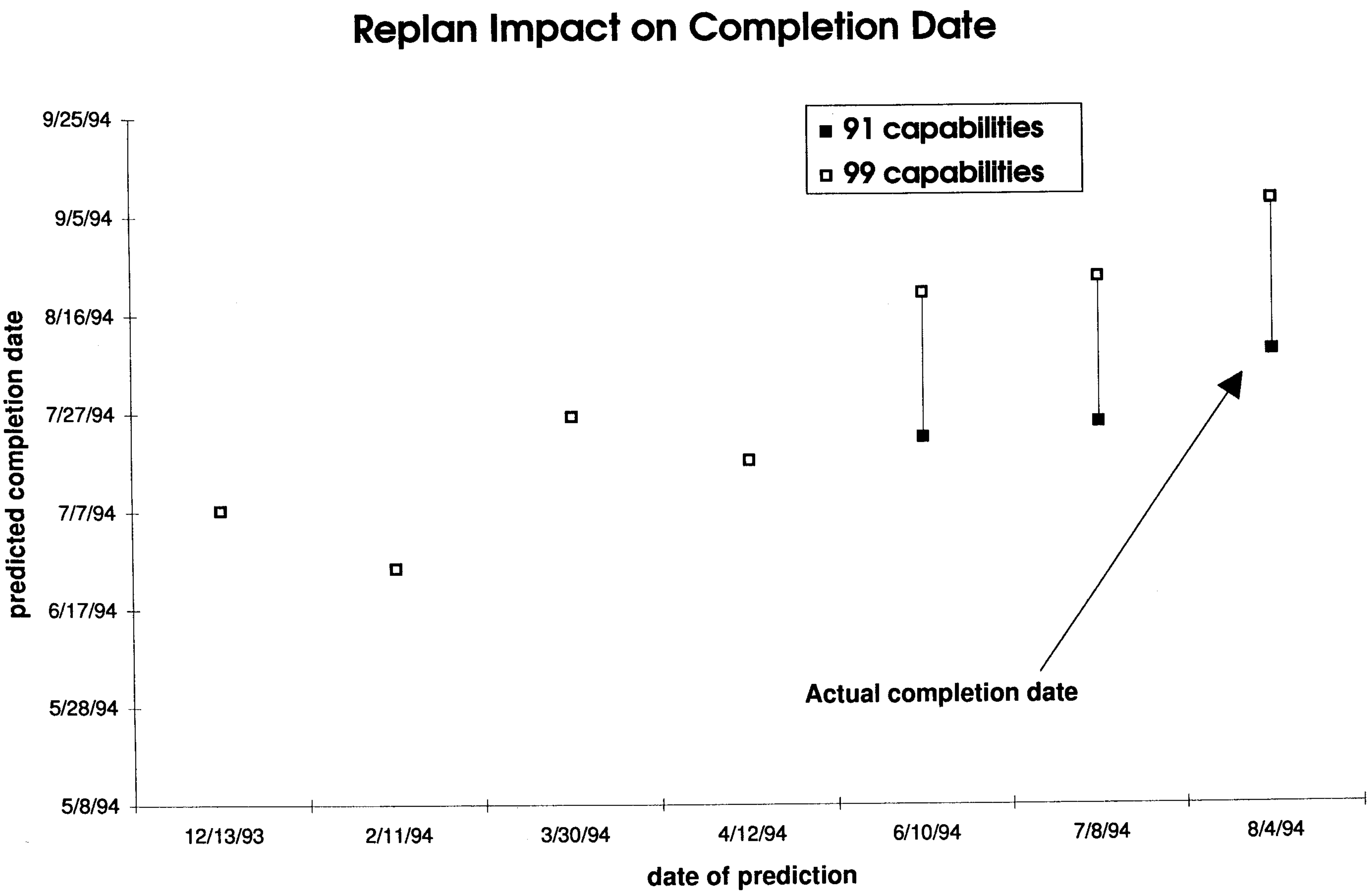

Replan Impact on Completion Date
■ 91 capabilities
□ 99 capabilities
predicted completion date
9/25/94
9/5/94
8/16/94
7/27/94
7/7/94
6/17/94
5/28/94
5/8/94
12/13/93
2/11/94
3/30/94
4/12/94
6/10/94
7/8/94
8/4/94
date of prediction
Actual completion date

Figure C-2

Distribution of Change in Time

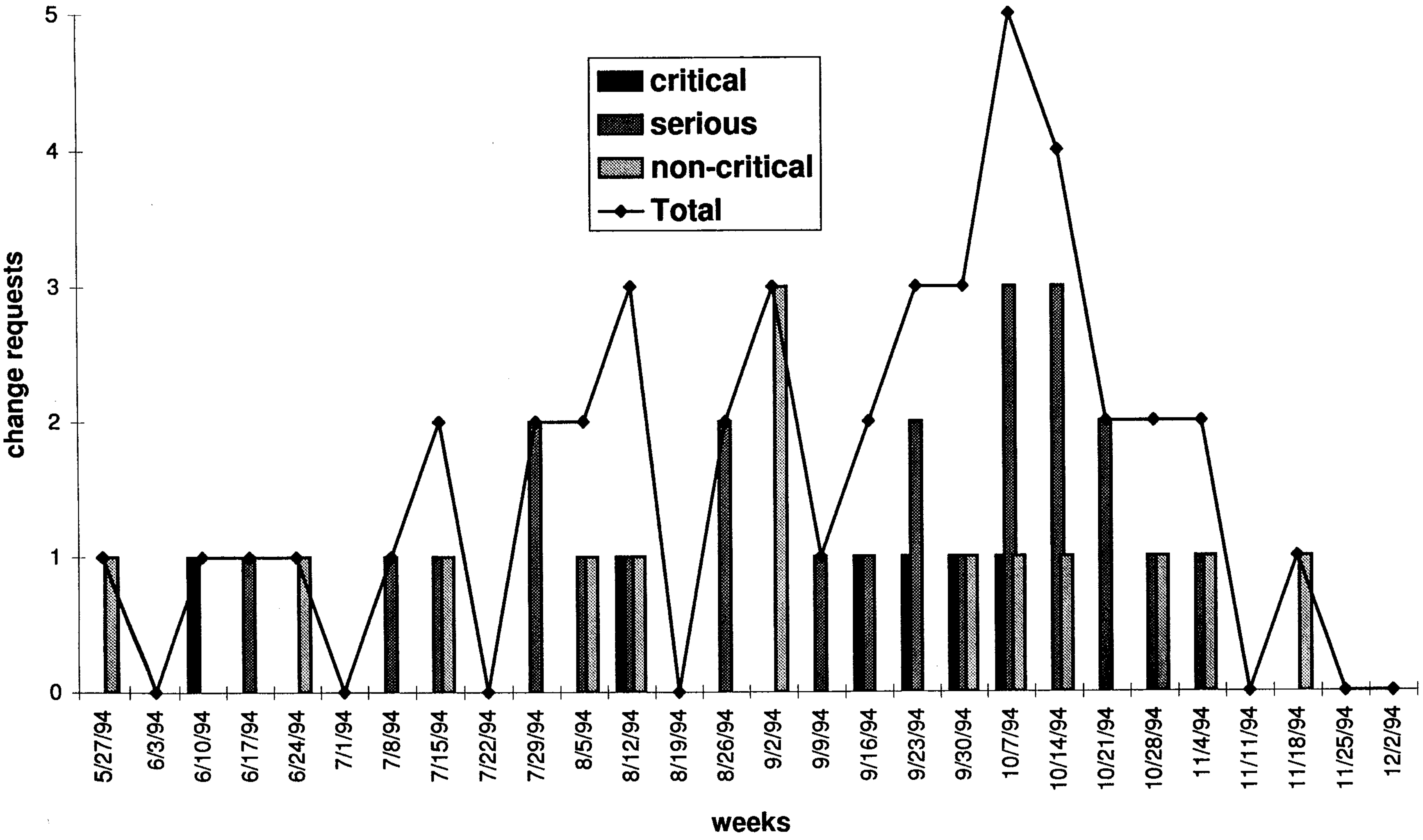


Figure C-3

Incident Report Arrival Rate

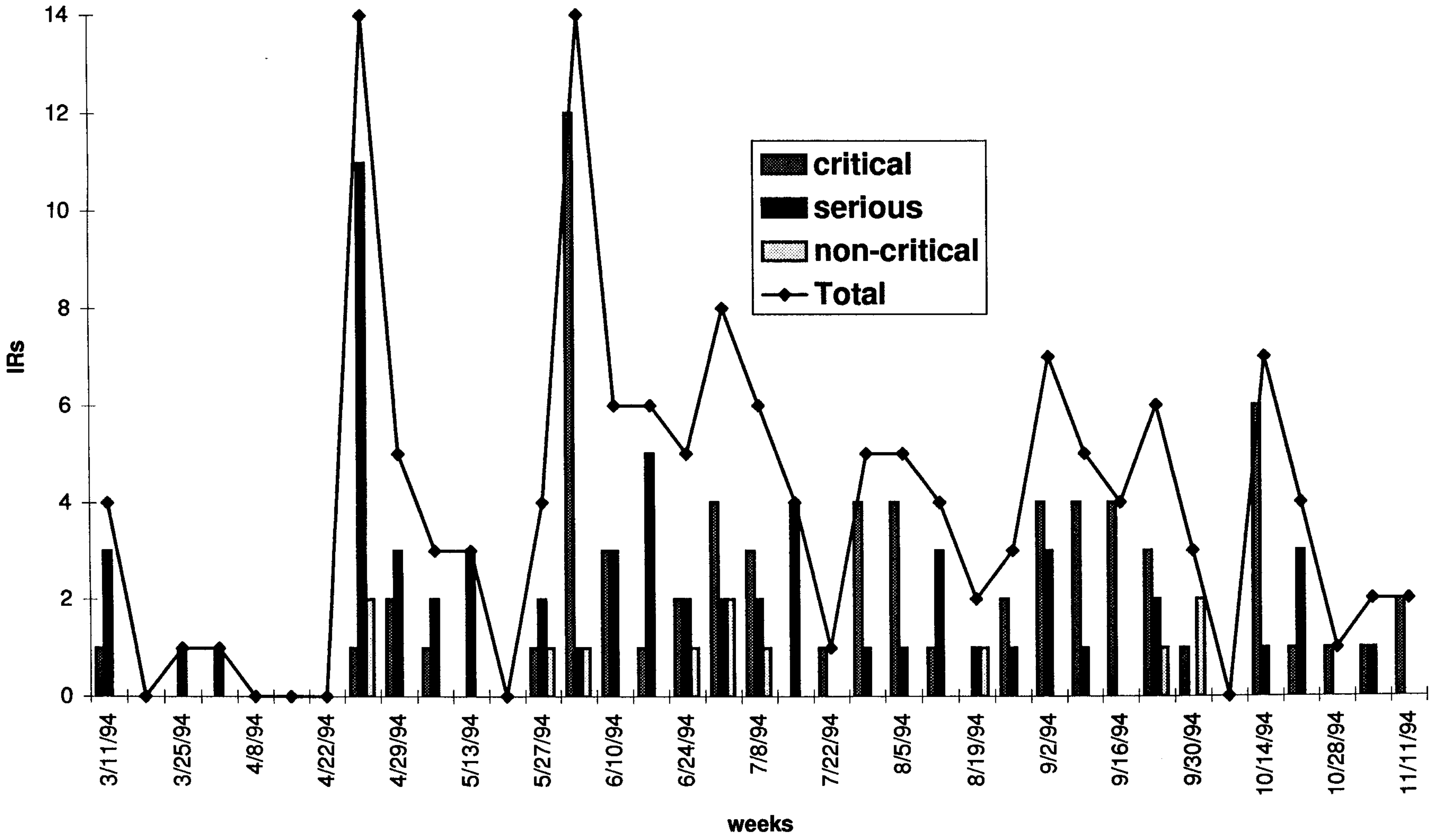


Figure D-1

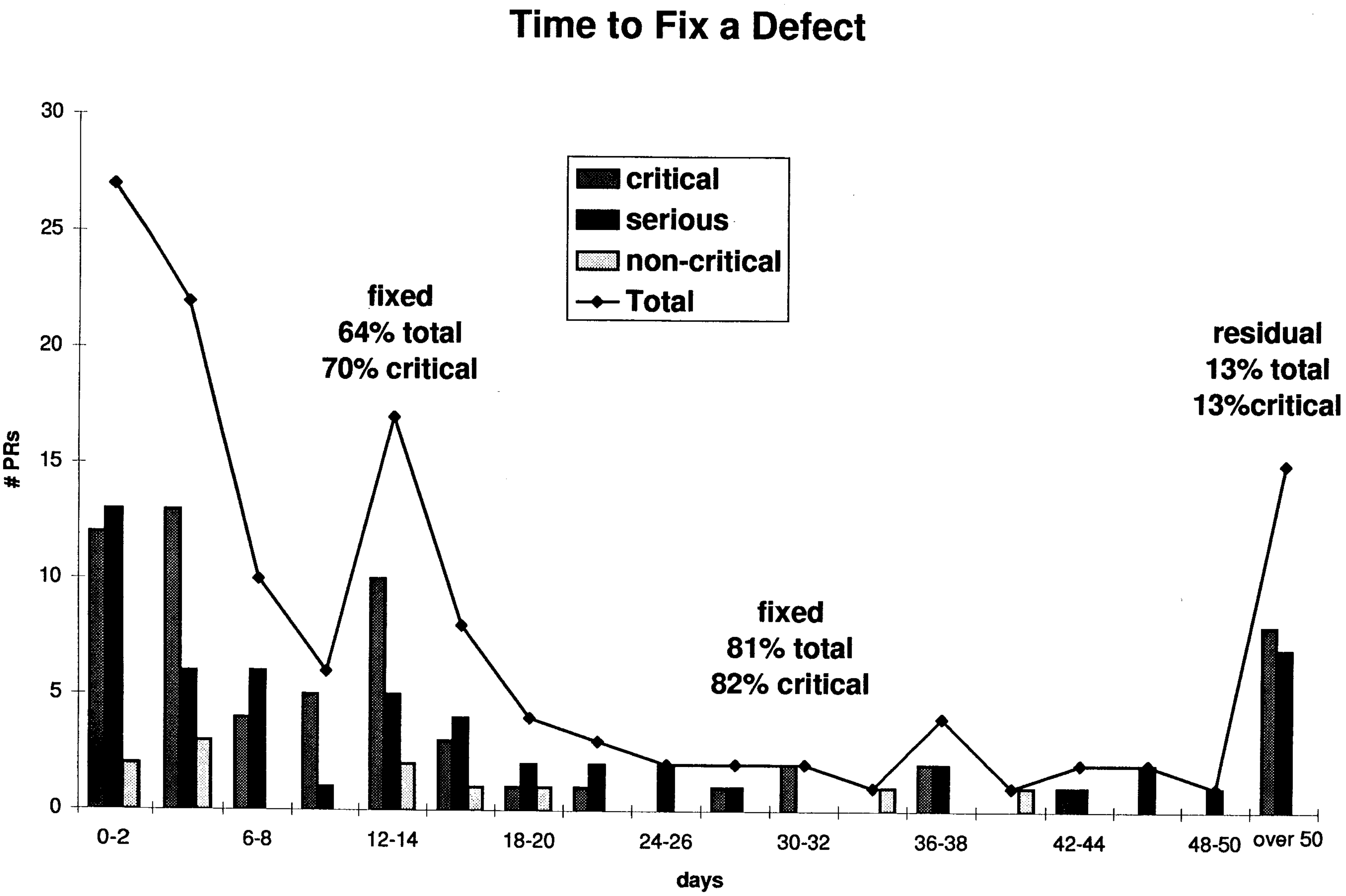

Time to Fix a Defect
critical
serious
non-critical
Total
fixed
64% total
70% critical
fixed
81% total
82% critical
residual
13% total
13%critical
# PRs
30
25
20
15
10
5
0
0-2
6-8
12-14
18-20
24-26
30-32
36-38
42-44
48-50
over 50
days

Figure D-2

Defects Discovered in Inspections by Type

Up to 2/18/94

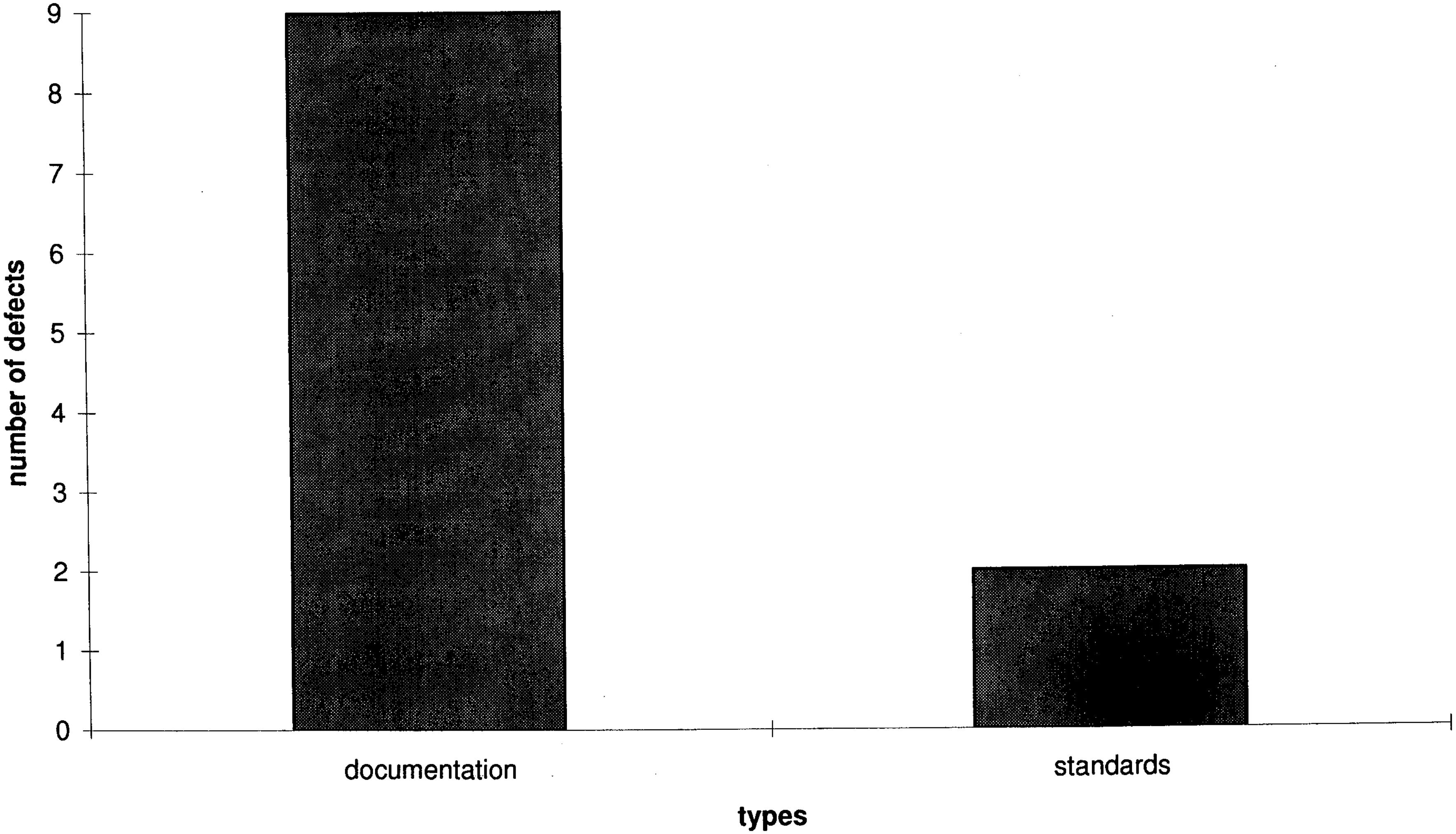


Figure D-3

Problem Severity Factor and Open Incident Reports

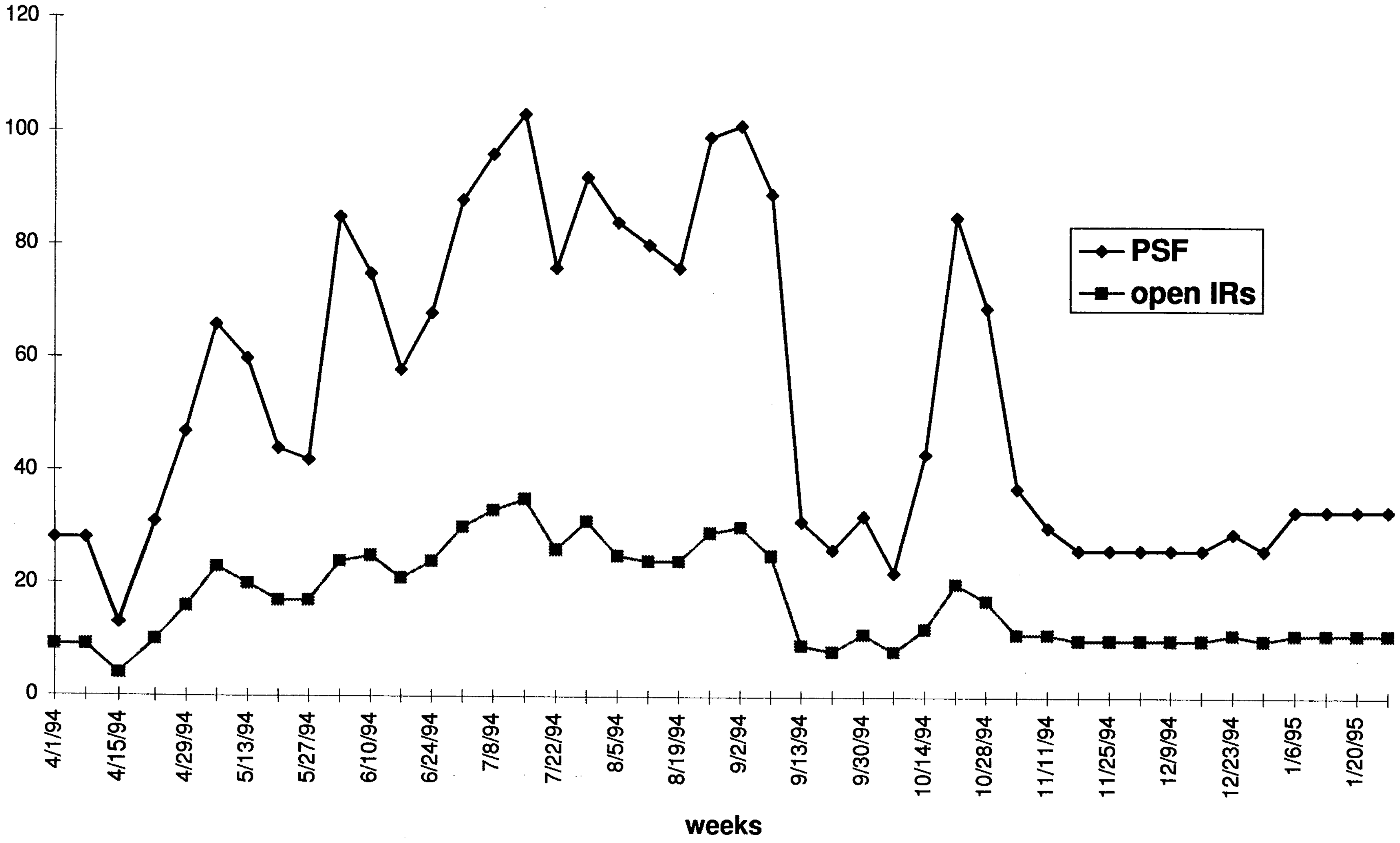


Figure D-4

SDRR: Problem Severity Factor by Tool

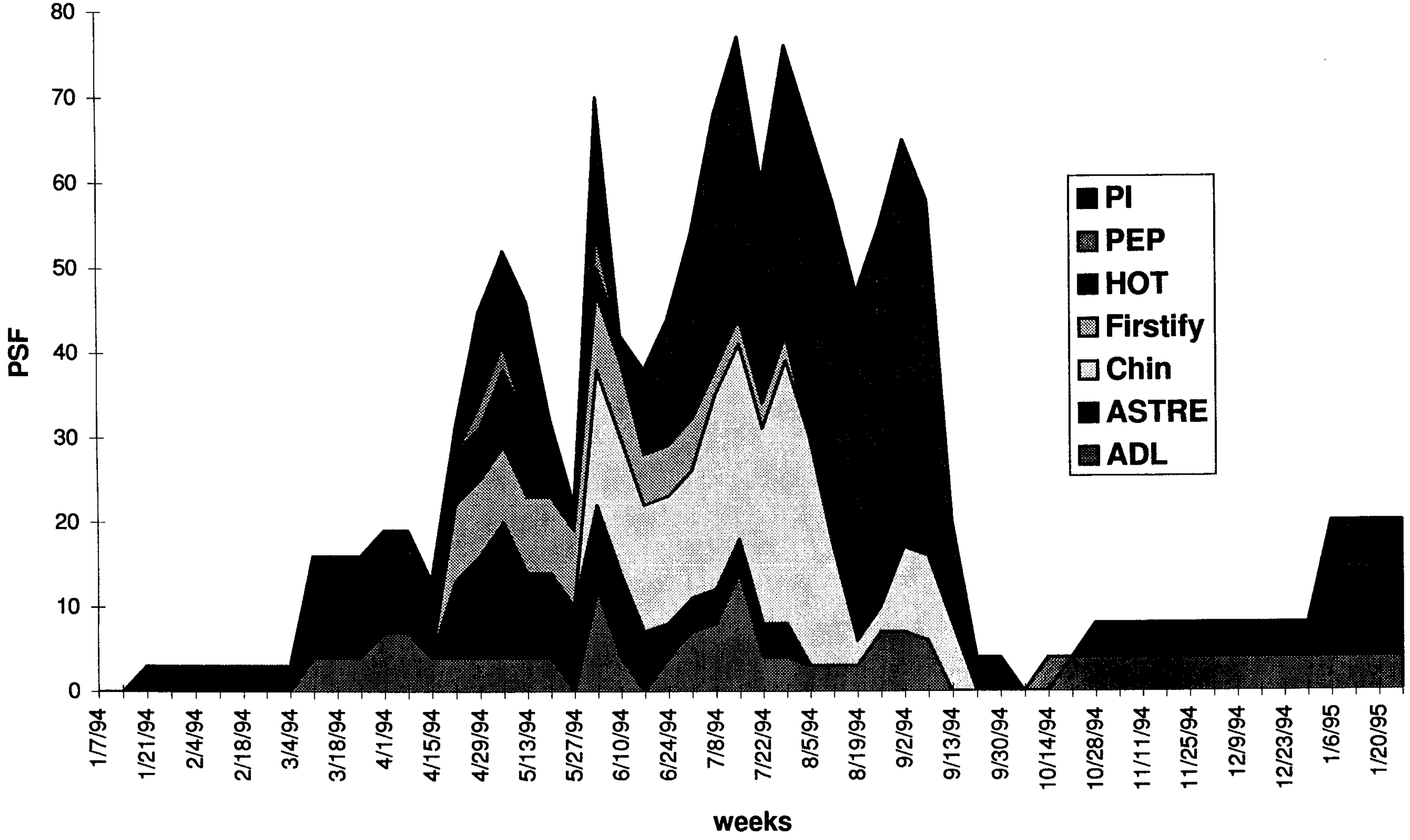


Figure D-5

MSL: Problem Severity Factor

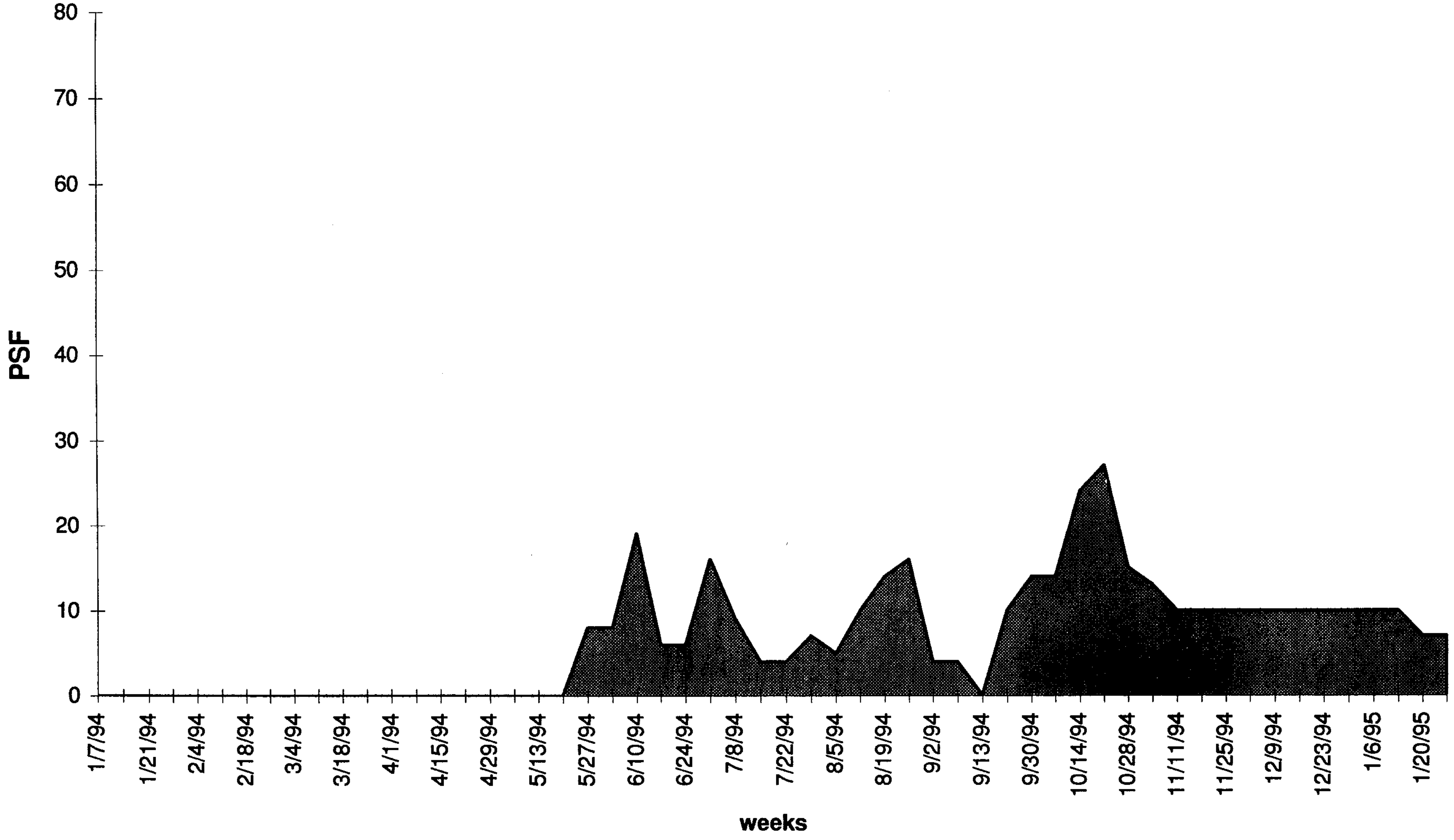
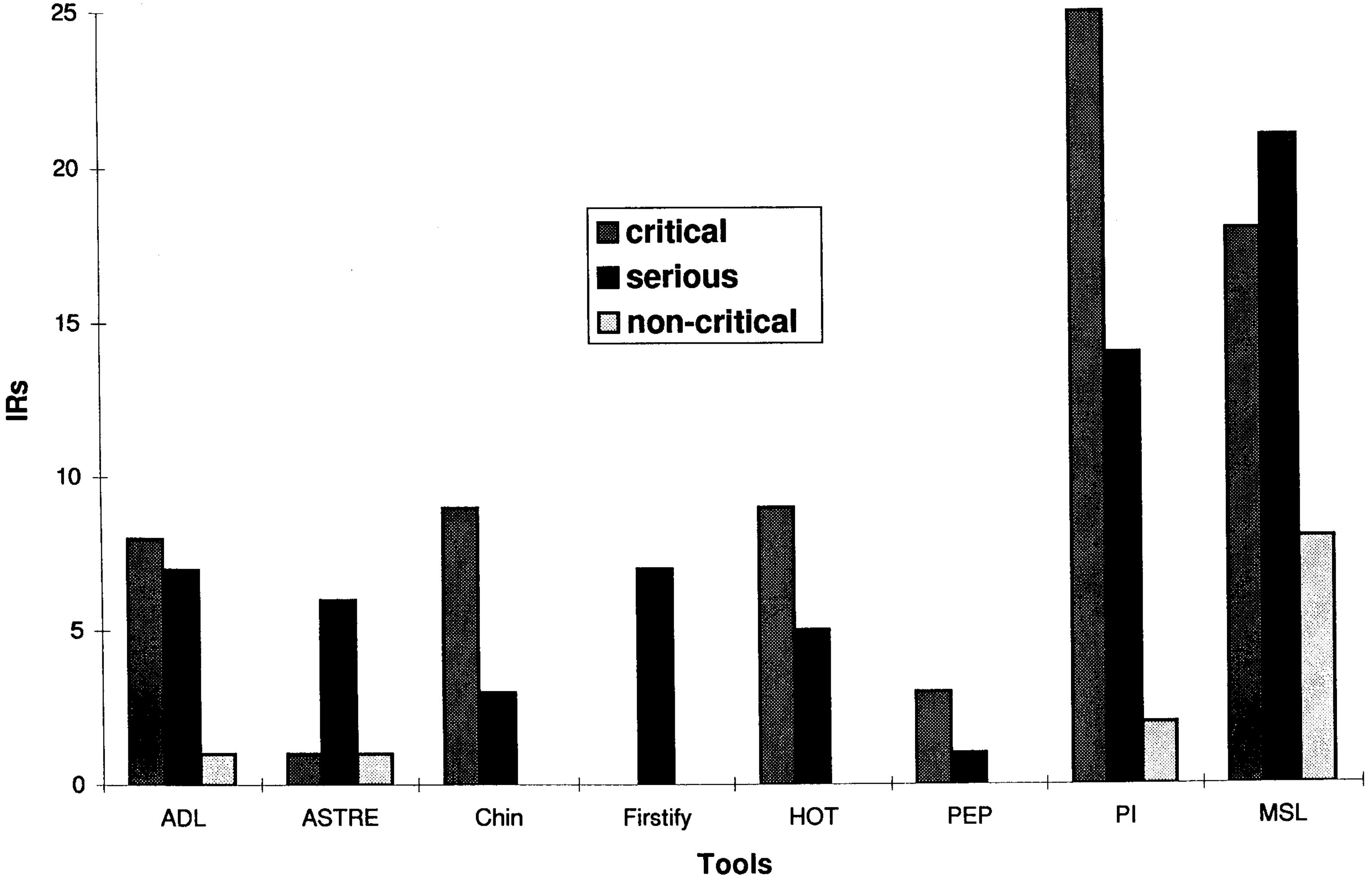


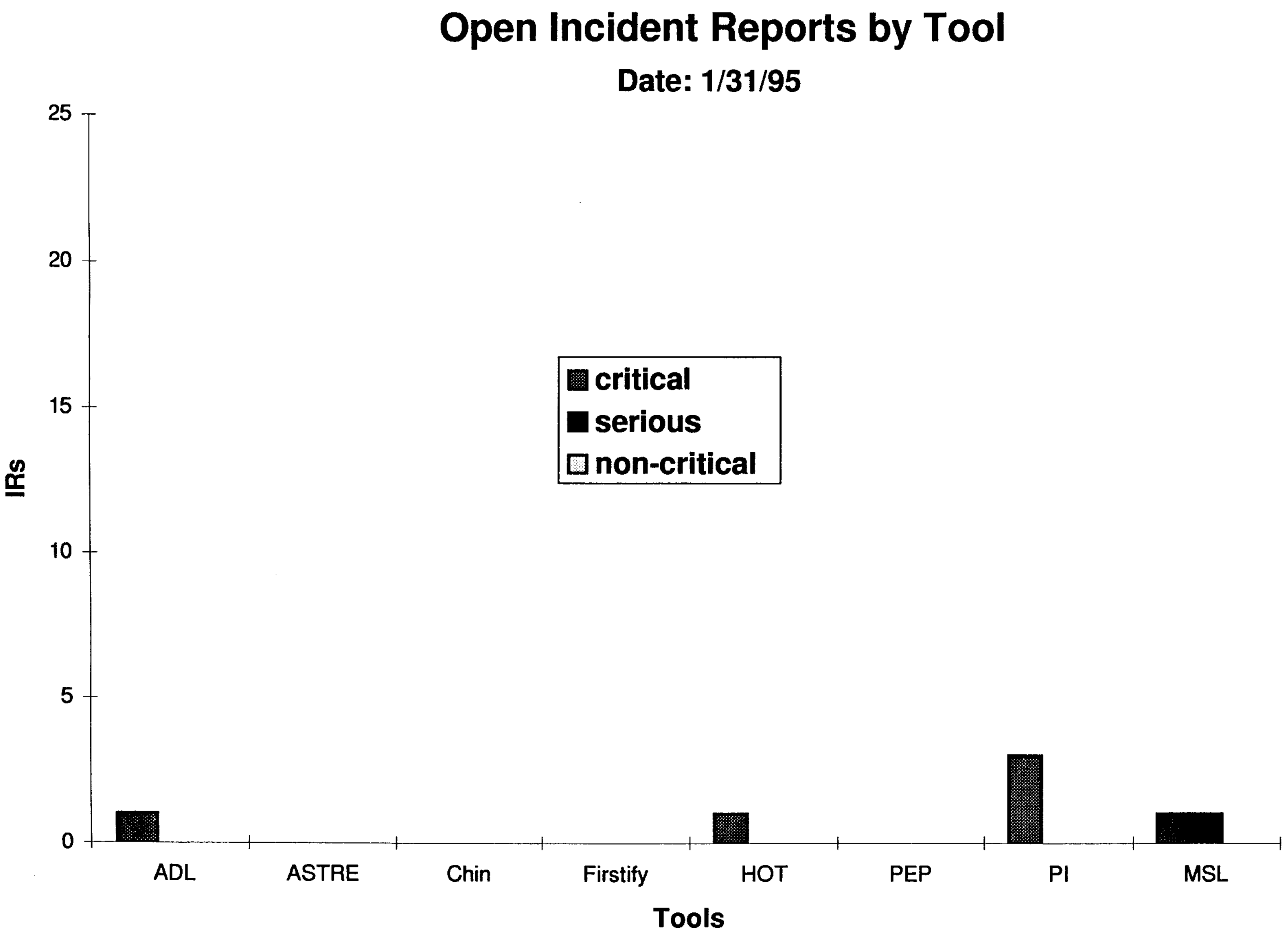
Figure D-6



Total Number of Incident Reports by Tool

Date: 1/31/95

Figure D-7



Open Incident Reports by Tool

Date: 1/31/95

Figure D-8

Cumulative Defects, Project

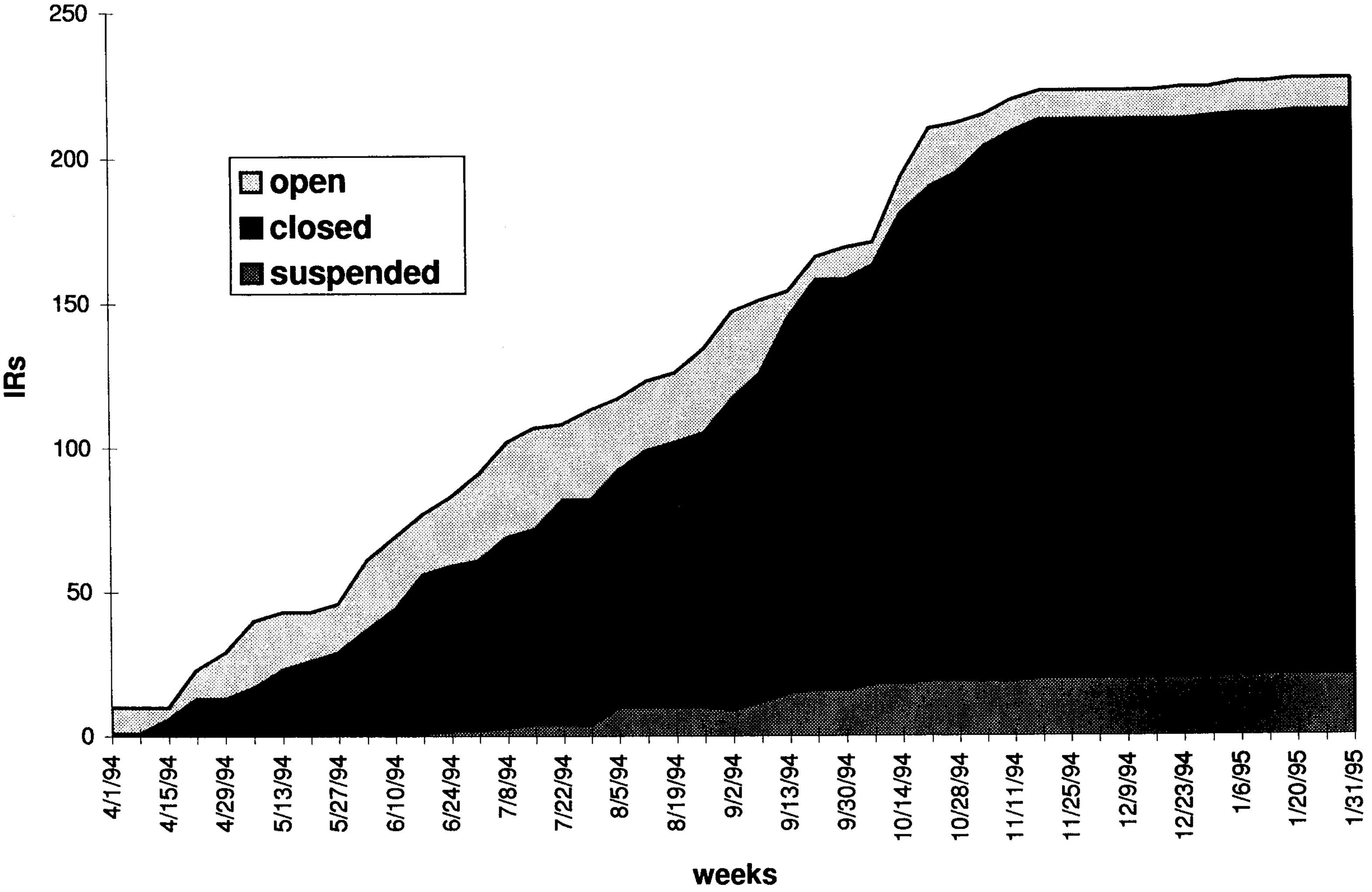


Figure D-9

SDRR Development Defect Rate by Tool

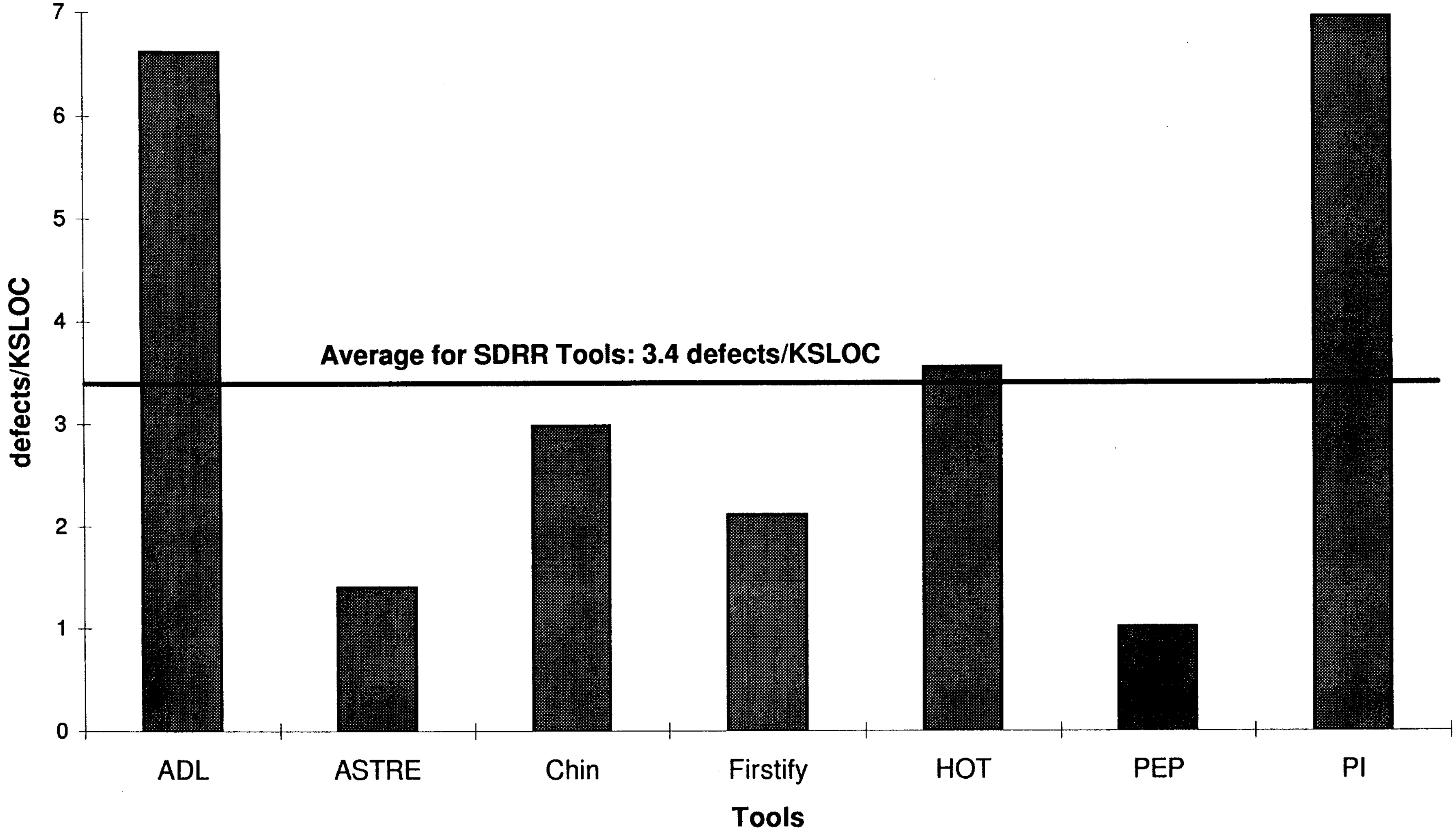


Figure D-10

Project Development Defect Rate by Tool

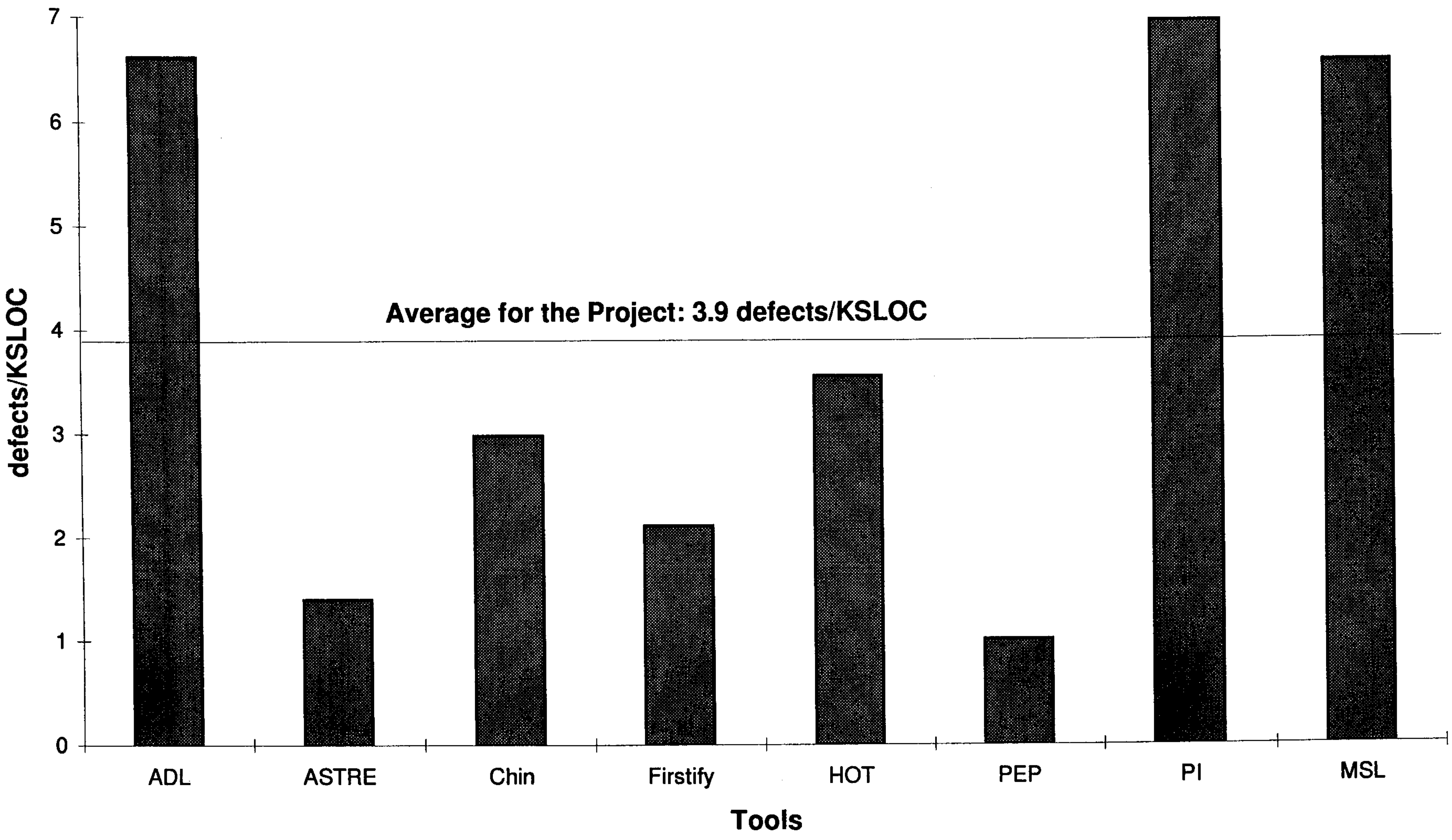


Figure D-11

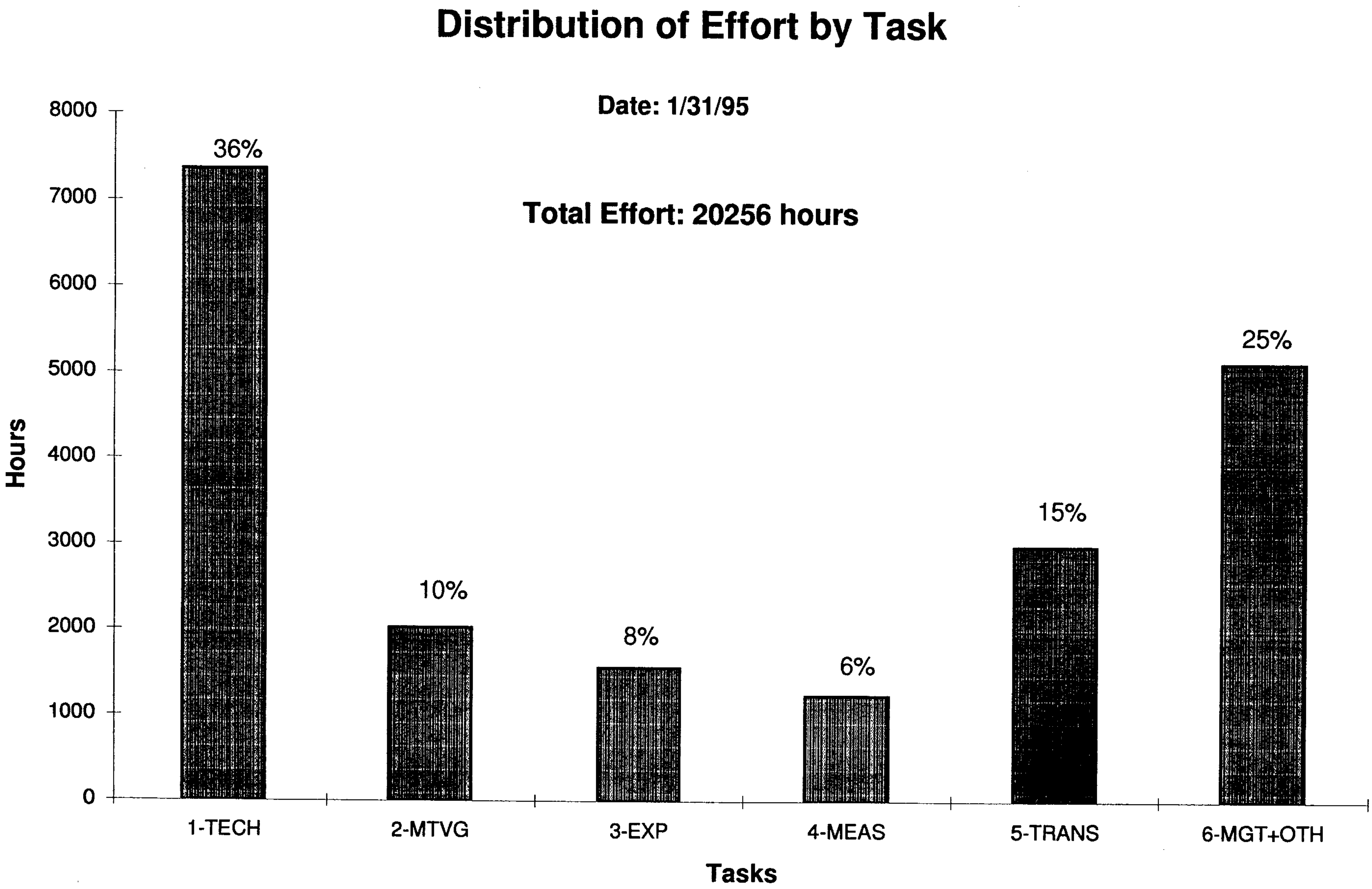

Distribution of Effort by Task
Date: 1/31/95
Total Effort: 20256 hours
Hours
8000
7000
6000
5000
4000
3000
2000
1000
0
36%
10%
8%
6%
15%
25%
1-TECH
2-MTVG
3-EXP
4-MEAS
5-TRANS
6-MGT+OTH
Tasks

Figure E-1

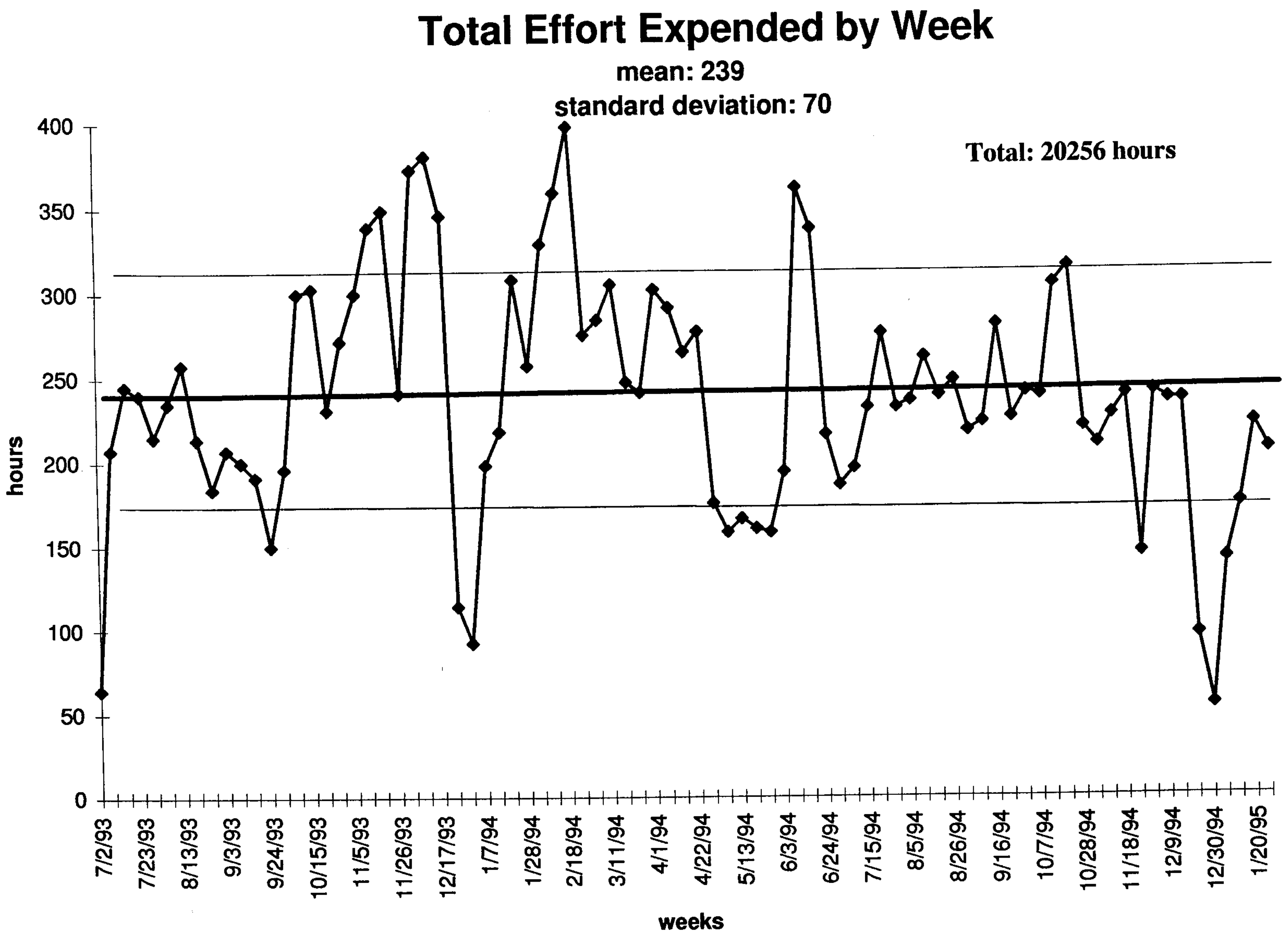

Total Effort Expended by Week
mean: 239
standard deviation: 70
Total: 20256 hours
hours
400
350
300
250
200
150
100
50
0
7/2/93
7/23/93
8/13/93
9/3/93
9/24/93
10/15/93
11/5/93
11/26/93
12/17/93
1/7/94
1/28/94
2/18/94
3/11/94
4/1/94
4/22/94
5/13/94
6/3/94
6/24/94
7/15/94
8/5/94
8/26/94
9/16/94
10/7/94
10/28/94
11/18/94
12/9/94
12/30/94
1/20/95
weeks

Figure E-2

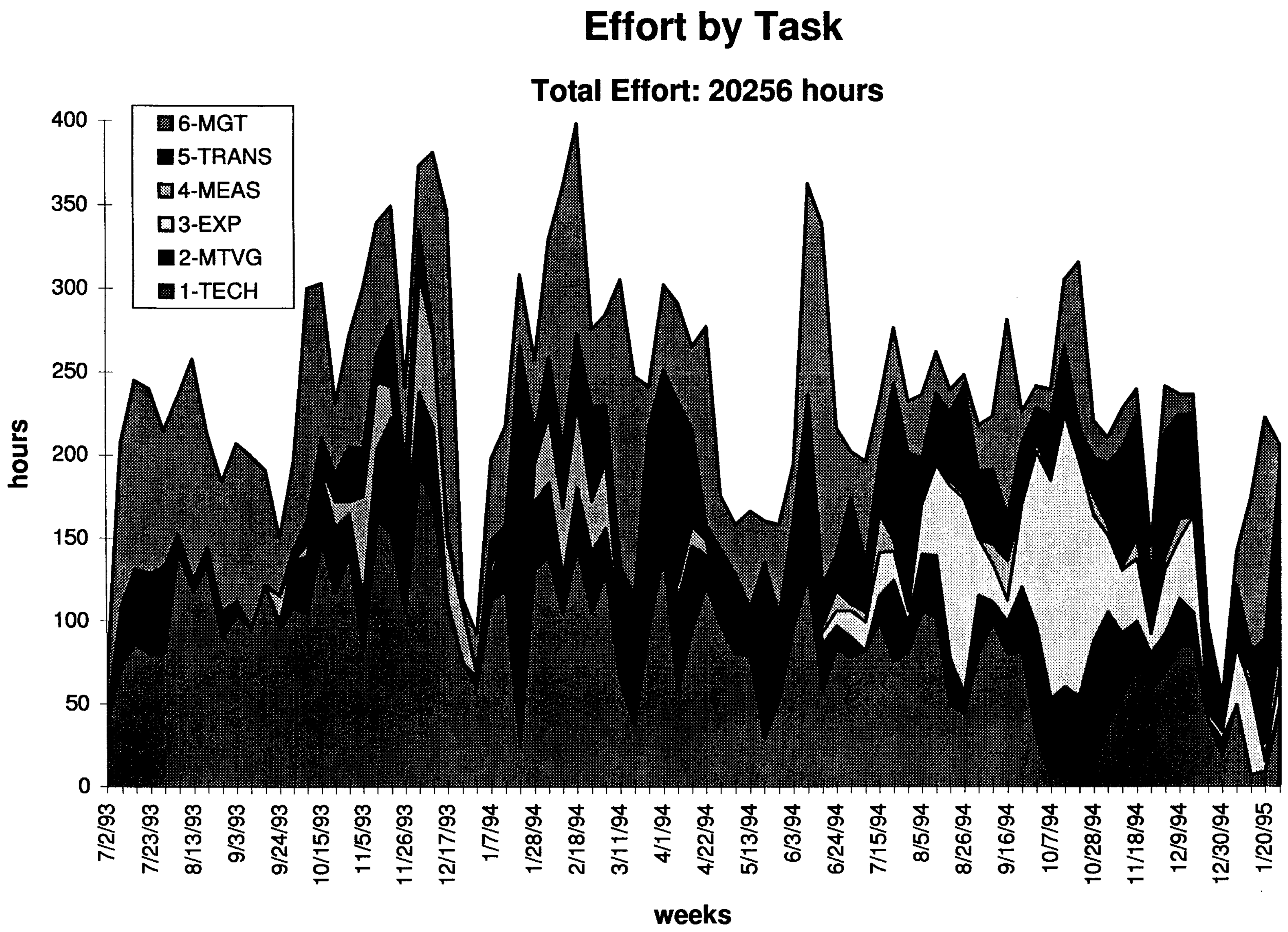
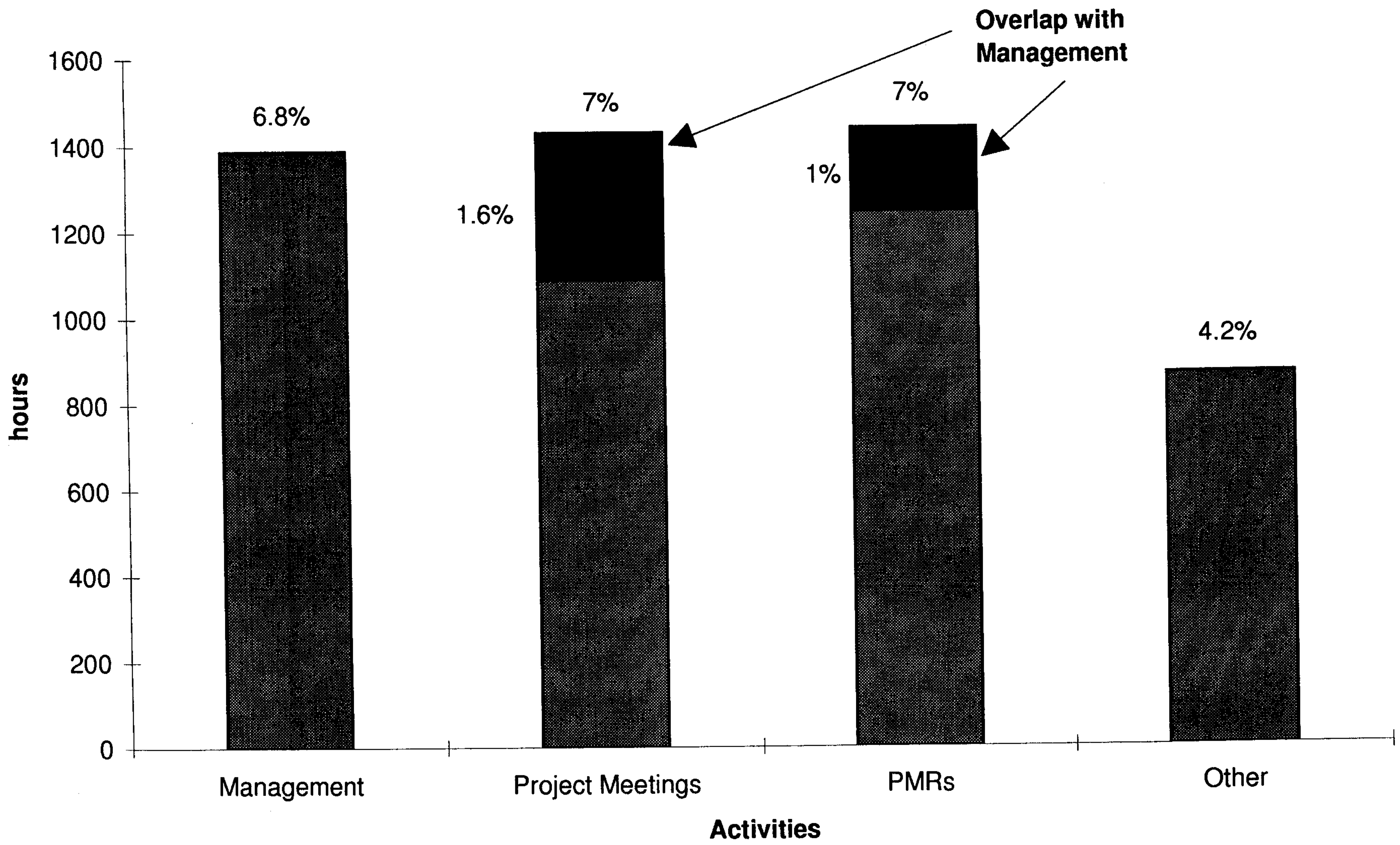

Effort by Task
Total Effort: 20256 hours
6-MGT
5-TRANS
4-MEAS
3-EXP
2-MTVG
1-TECH
hours
400
350
300
250
200
150
100
50
0
7/2/93
7/23/93
8/13/93
9/3/93
9/24/93
10/15/93
11/5/93
11/26/93
12/17/93
1/7/94
1/28/94
2/18/94
3/11/94
4/1/94
4/22/94
5/13/94
6/3/94
6/24/94
7/15/94
8/5/94
8/26/94
9/16/94
10/7/94
10/28/94
11/18/94
12/9/94
12/30/94
1/20/95
weeks

Figure E-3



Distribution of Effort, Task 6, % of Total Project Effort

Figure E-4

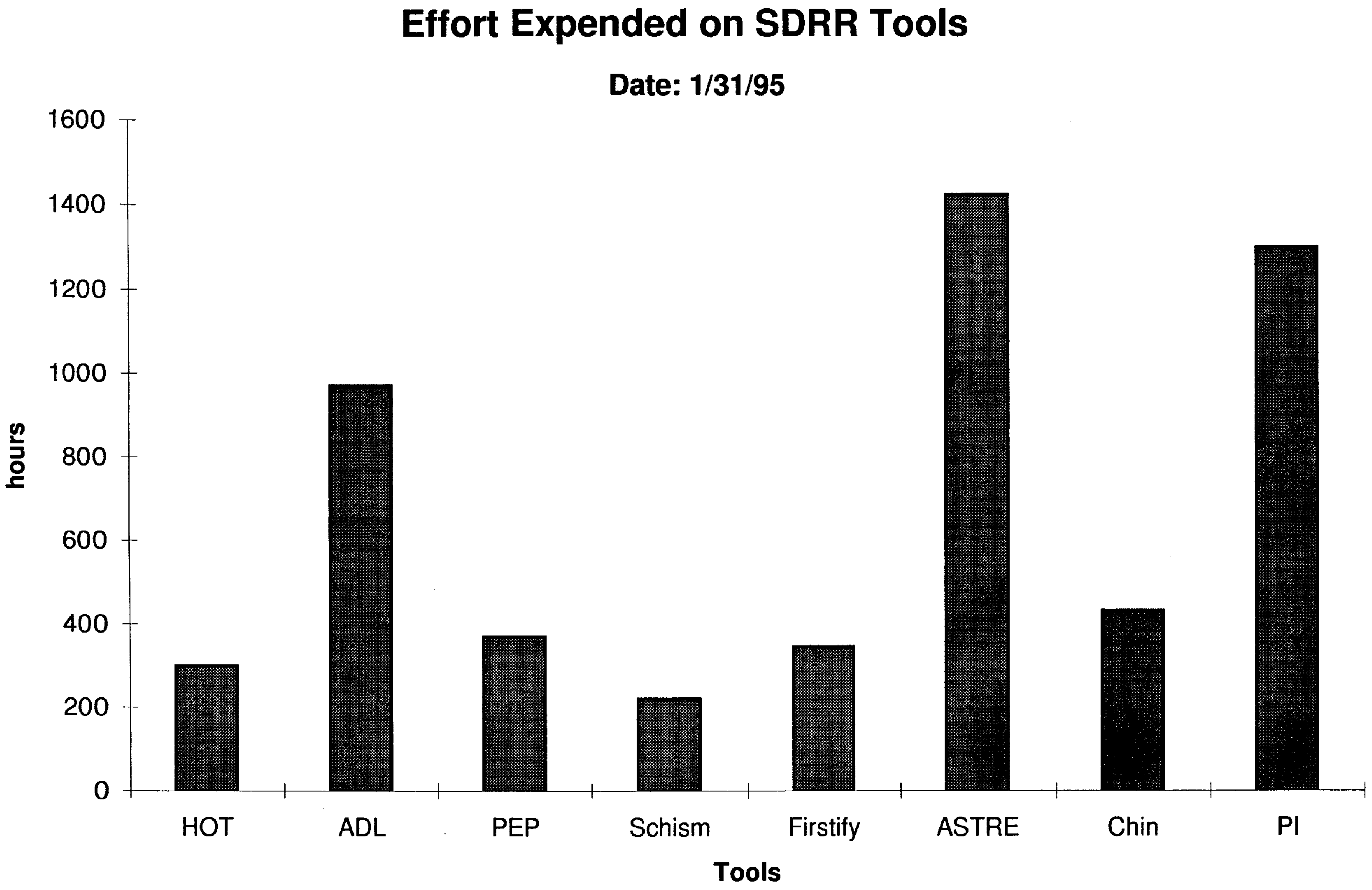

Effort Expended on SDRR Tools
Date: 1/31/95
hours
1600
1400
1200
1000
800
600
400
200
0
HOT
ADL
PEP
Schism
Firstify
ASTRE
Chin
PI
Tools

Figure E-5

Predicted Completion Date for Code Development

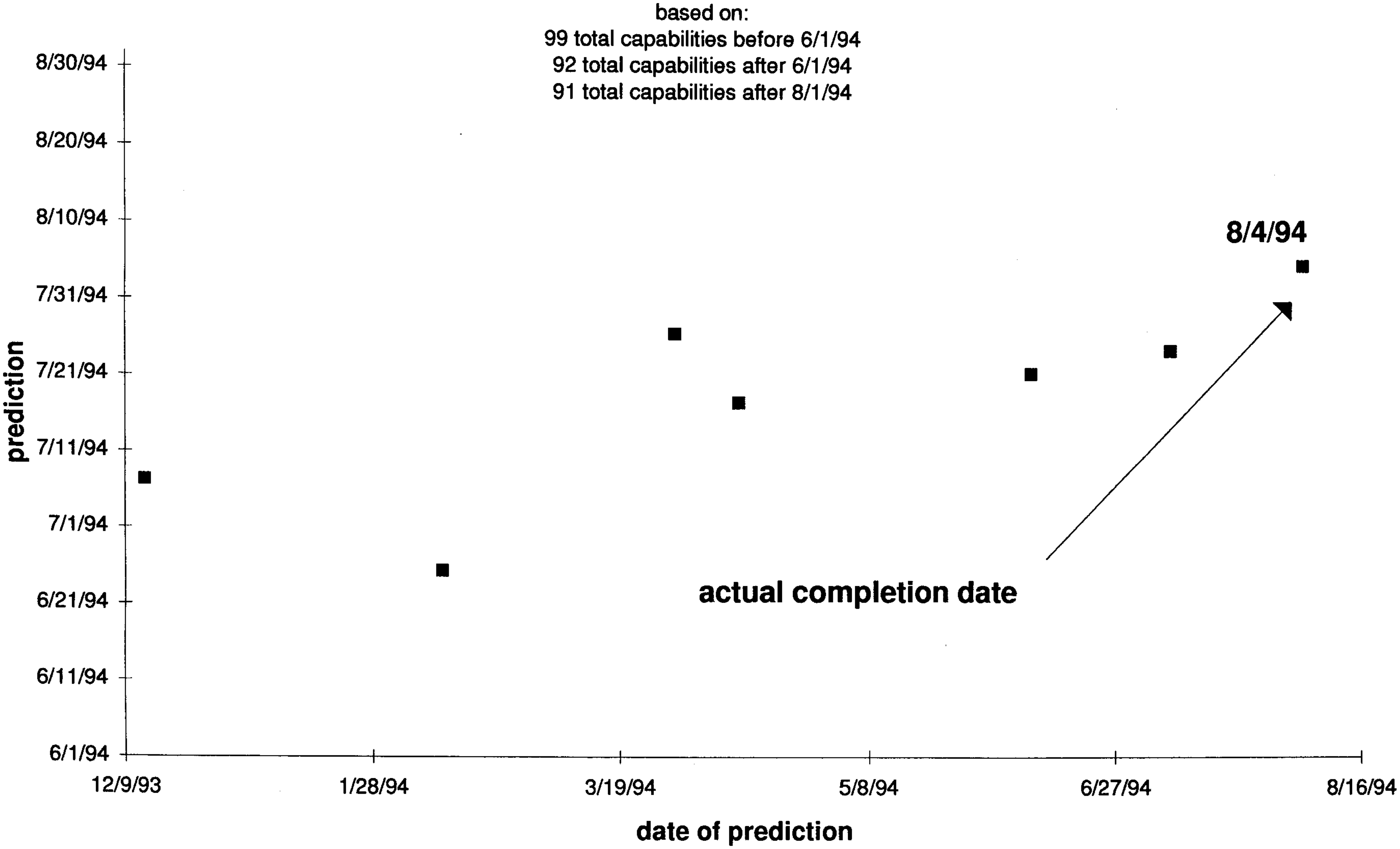


Figure P-1

Actual and Predicted SW Size, ELOC (ASTRE excluded)

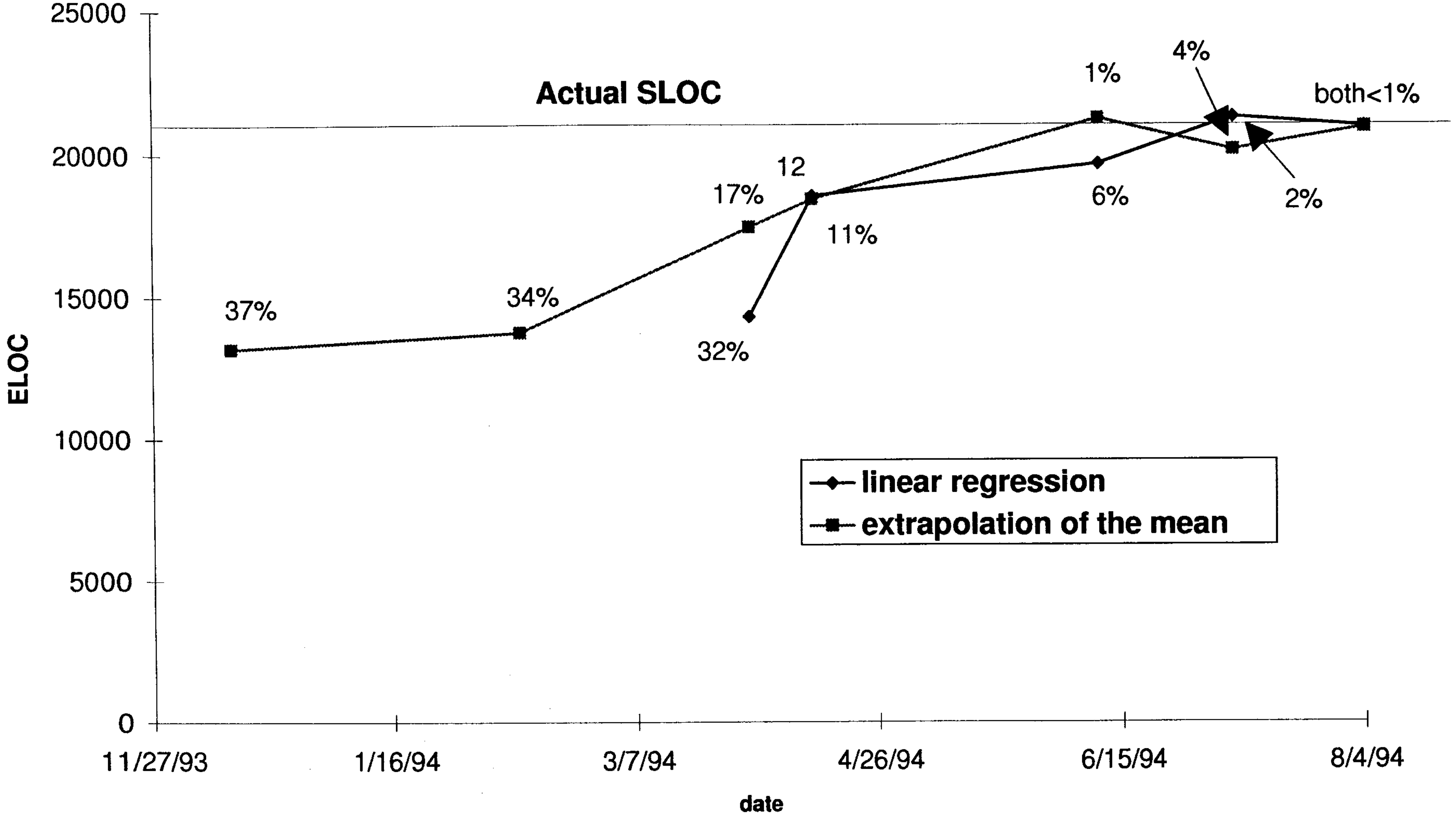


Figure P-2

Progress to Schedule and Predicted Completion Date

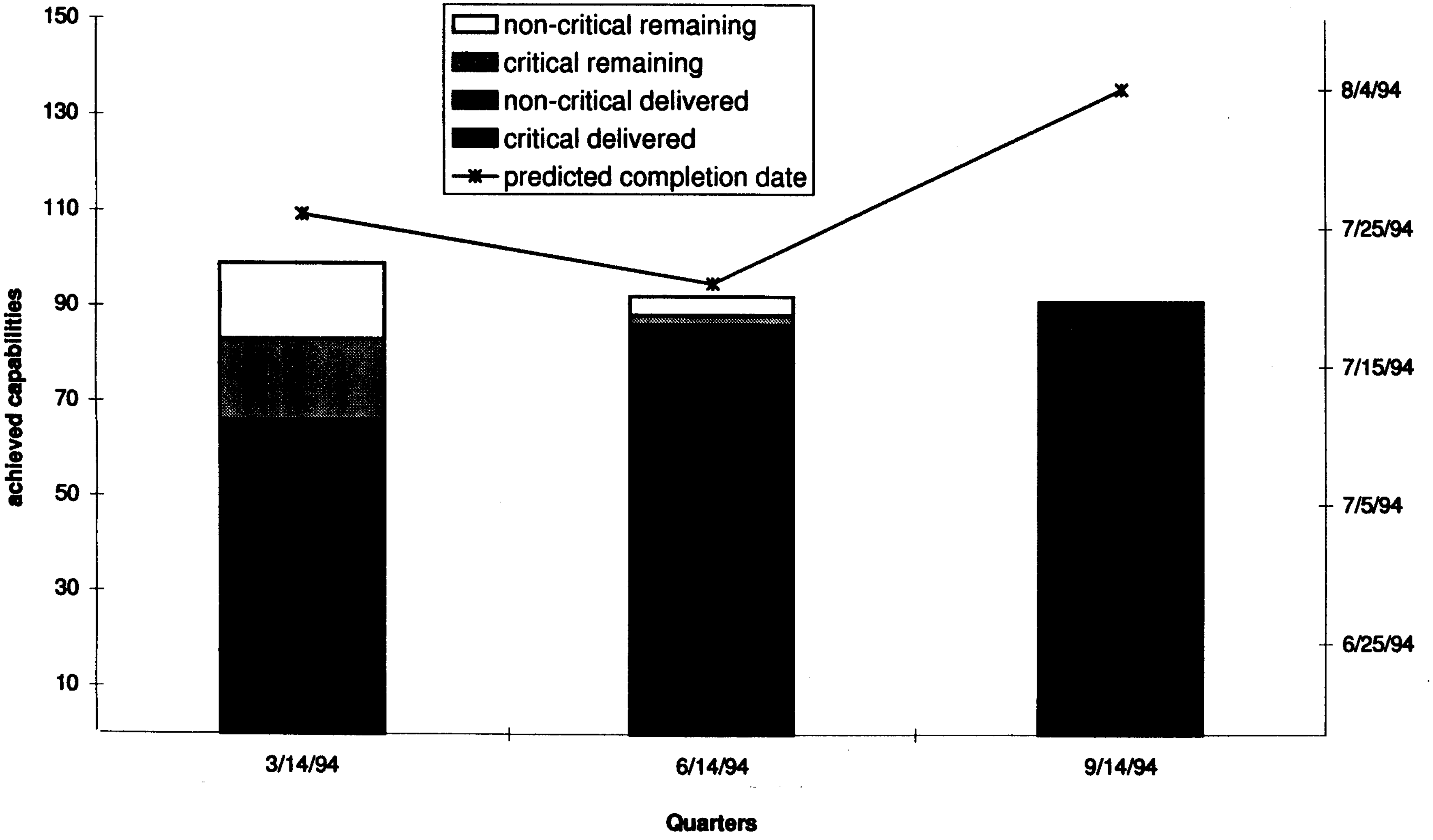


Figure P-3

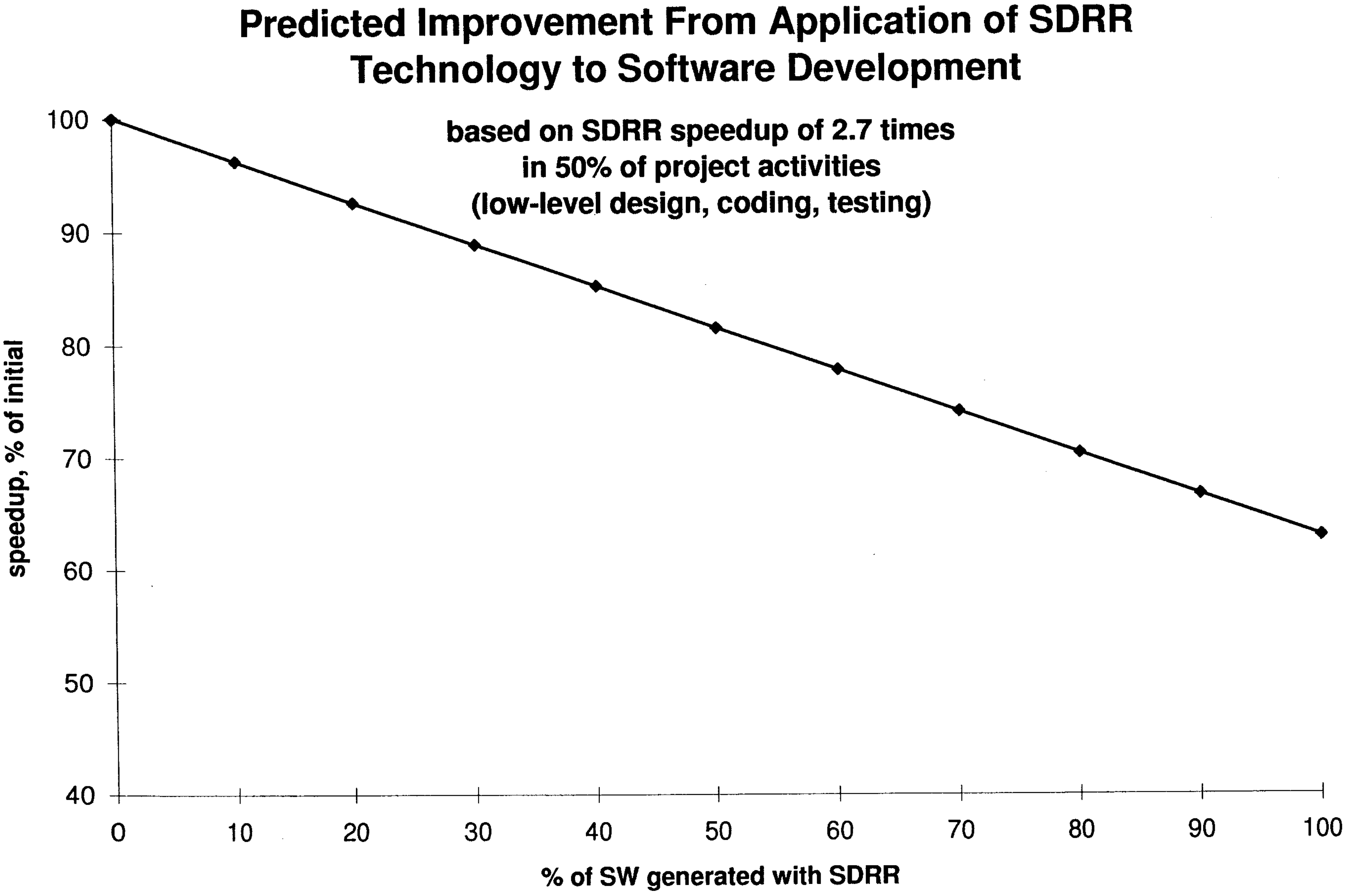

Predicted Improvement From Application of SDRR
Technology to Software Development
based on SDRR speedup of 2.7 times
in 50% of project activities
(low-level design, coding, testing)
speedup, % of initial
100
90
80
70
60
50
40
0
10
20
30
40
50
60
70
80
90
100
% of SW generated with SDRR

Figure R-1

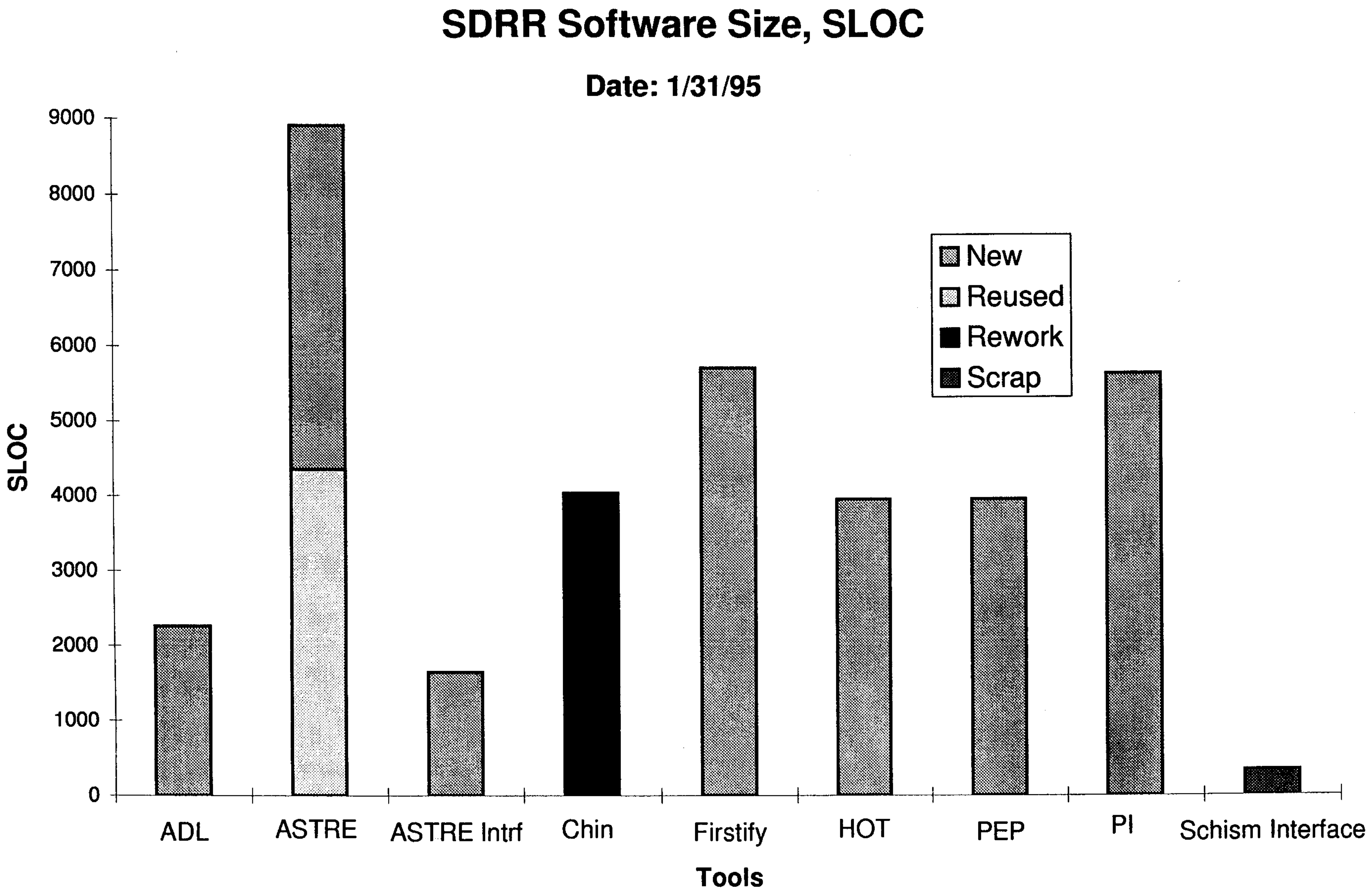

SDRR Software Size, SLOC
Date: 1/31/95
SLOC
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
New
Reused
Rework
Scrap
ADL
ASTRE
ASTRE Intrf
Chin
Firstify
HOT
PEP
PI
Schism Interface
Tools

Figure S-1

Code Accumulation, ELOC (ASTRE excluded)

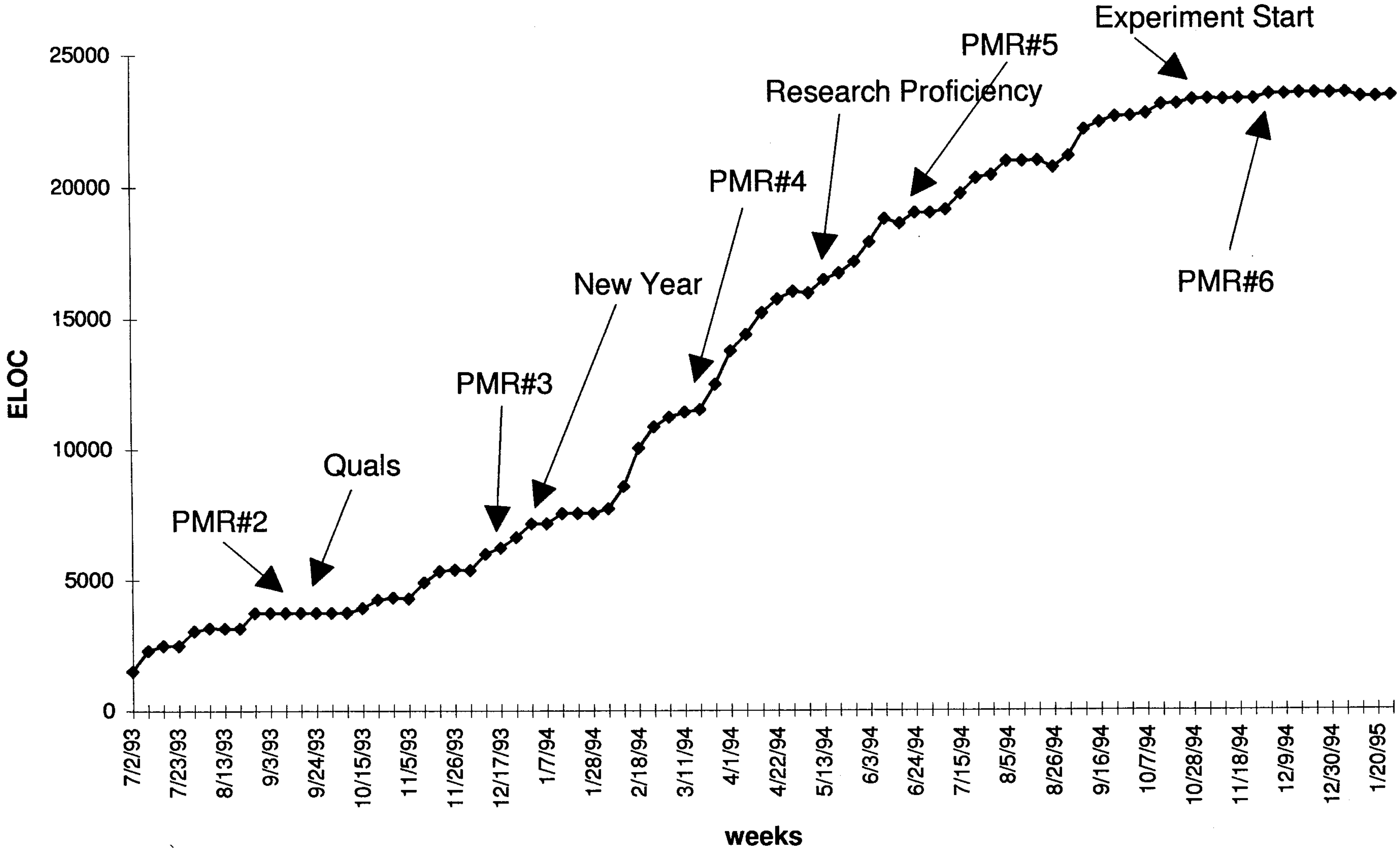


Figure S-2

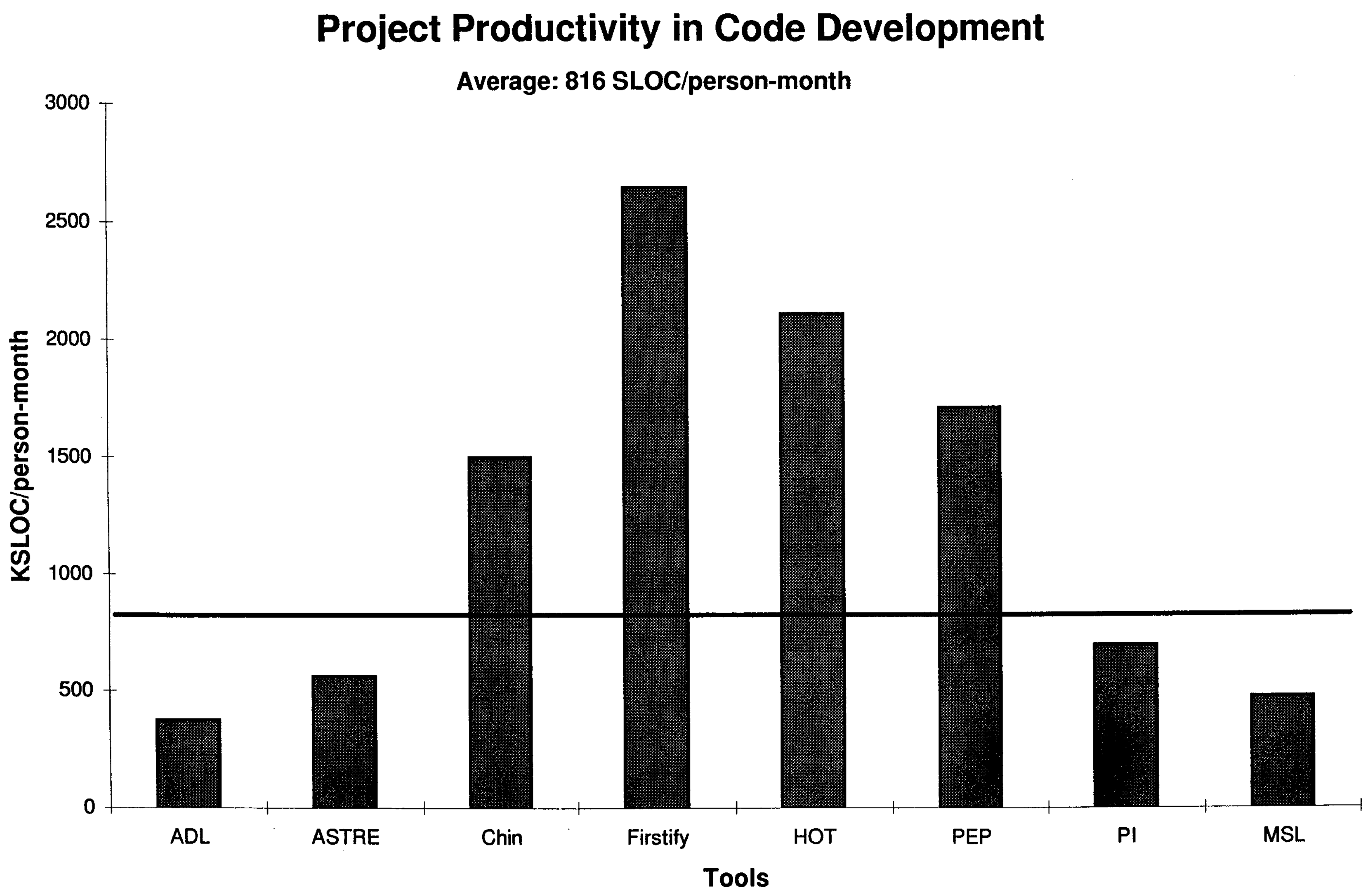

Project Productivity in Code Development
Average: 816 SLOC/person-month
KSLOC/person-month
3000
2500
2000
1500
1000
500
0
ADL
ASTRE
Chin
Firstify
HOT
PEP
PI
MSL
Tools

Figure S-3

Technology Exposure Through World-Wide Web

Date: 1/12/95

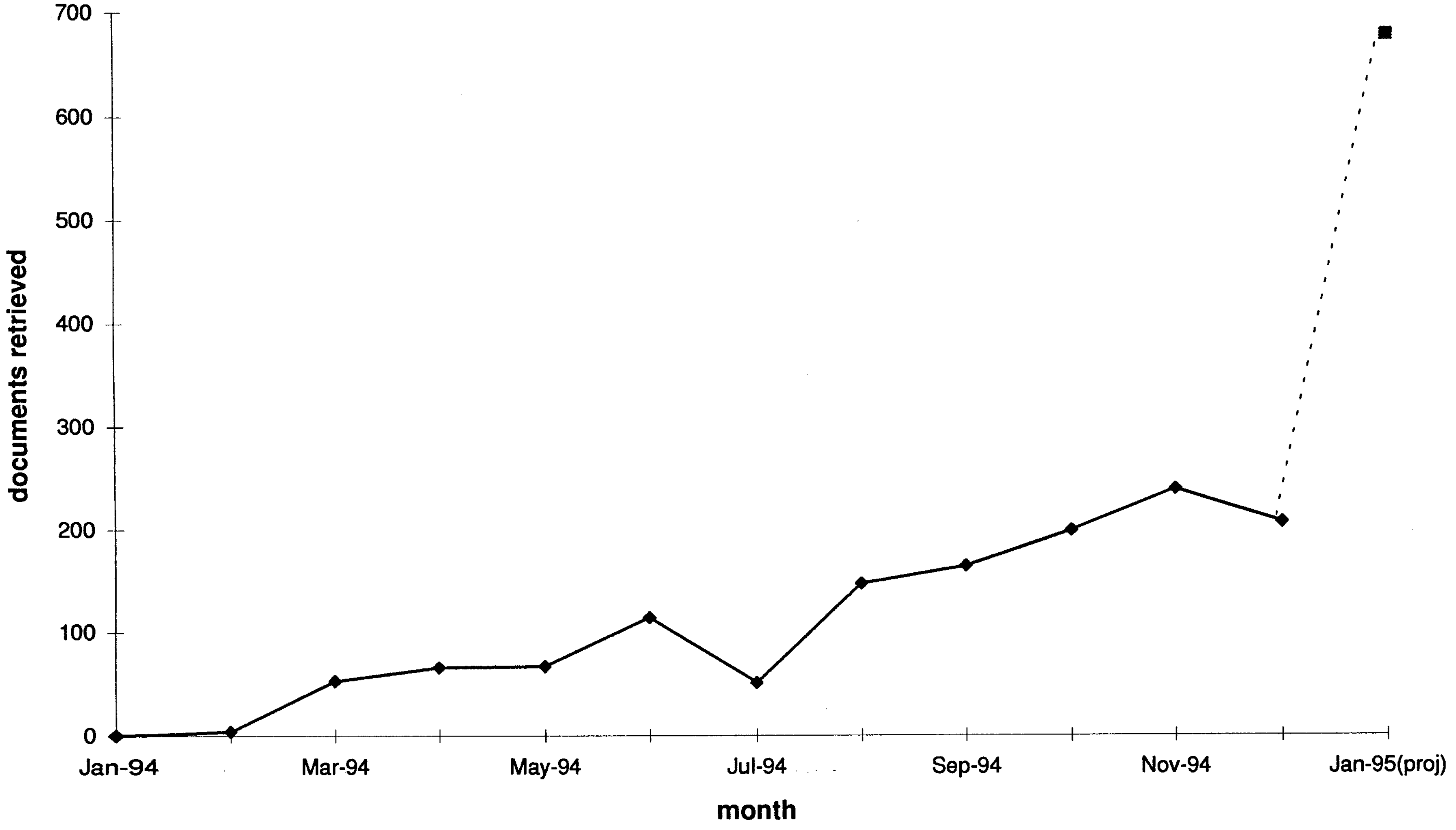


Figure T-1

Technology Transition Activities by Quarter

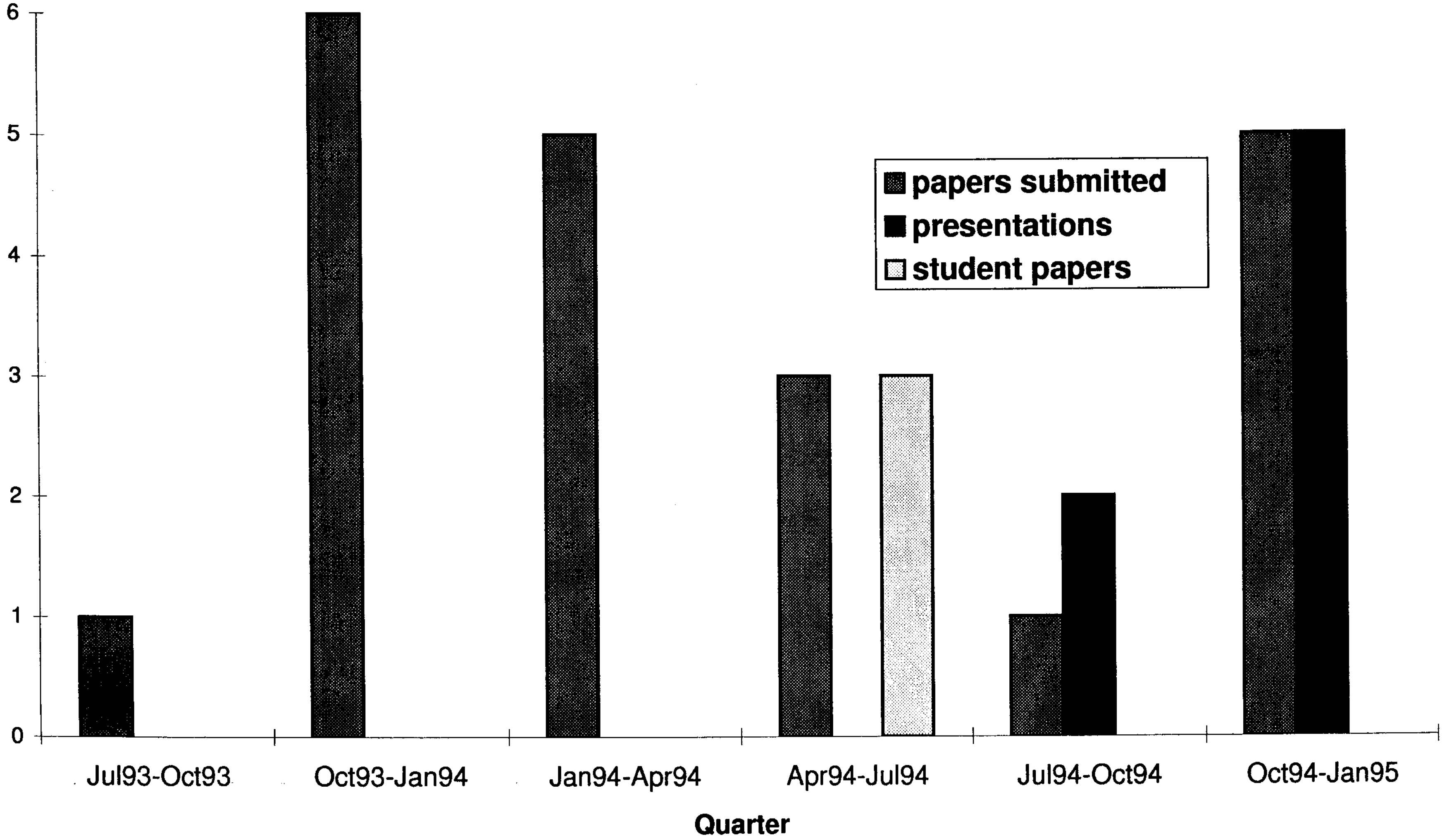


Figure T-2

Technology Exposure: Major WWW Accesses

November-December 1994

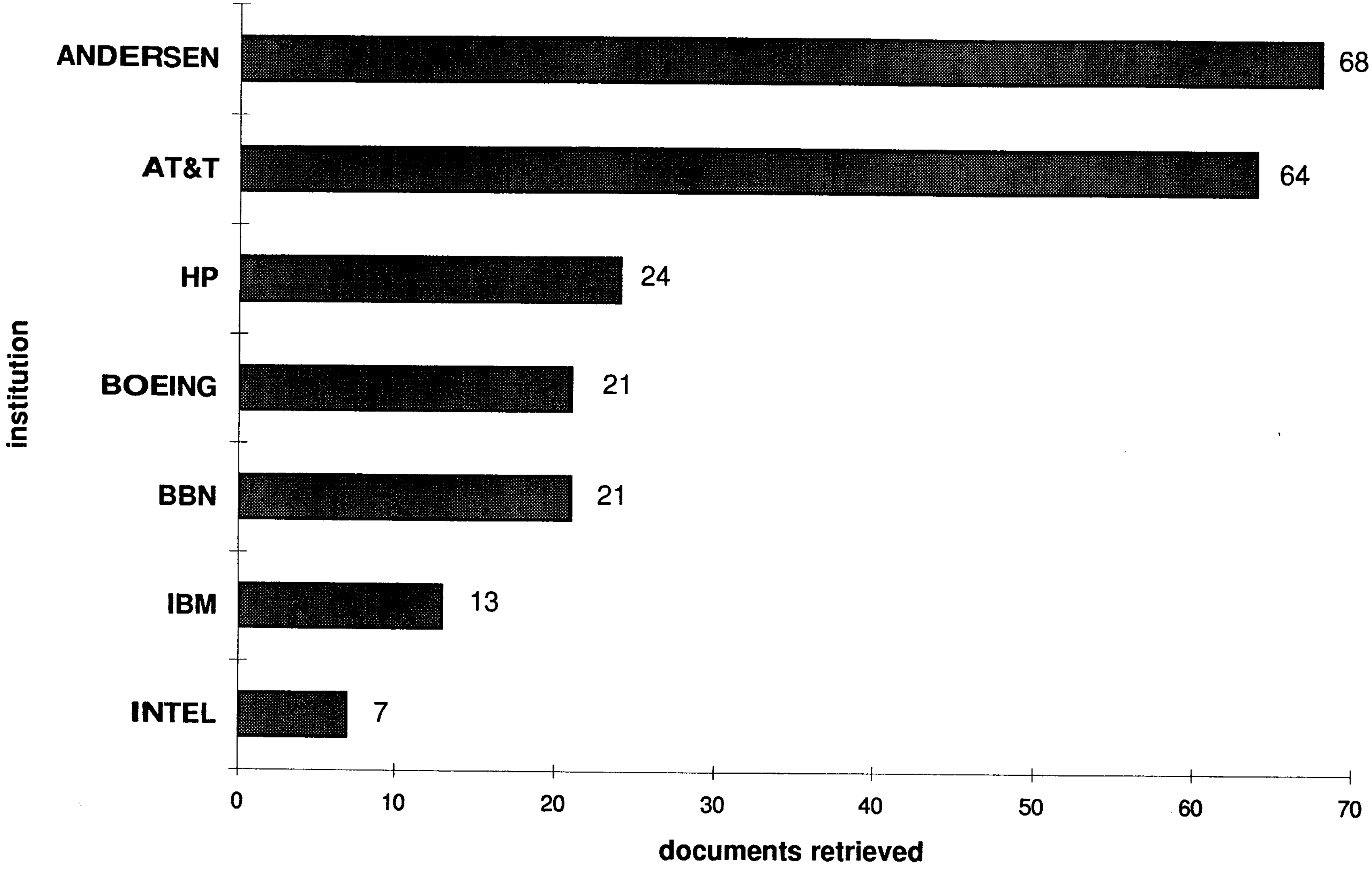


Figure T-3

Volume III
Design of the SDRR Pipeline

This architecture is intended to support reuse of the middle and final stages. The middle stage is, in principle, reused by every SDRR based generator. The final stage is reused whenever a target language is reused (i.e. all Ada-based generators will share the same final stage). Furthermore, the parameters to the final stage describing a particular environment can also be reused.

While this description is similar to that of a modern compiler, the choice of ADL as the output of the first stage presents research opportunities and challenges. It presents an opportunity because ADL's rich expressive power allows it to be used for a very high-level semantic description of the DSDL. It presents challenges because it is a fundamentally new type of functional programming language with new opportunities for optimization.

2 Design Goals

Many of the design goals were influenced by the context in which the system was developed. The system was built by a research group on a nineteen month contract. Furthermore, the contract included a three month validation experiment to be performed by independent contractors upon completion of the system. This dictated that the system be built in fifteen months (leaving one month to analyze and report the experiment results).

Many tools in the proposed system represented undemonstrated technology; concepts which were only captured on paper or in very crude prototypes.

In response to this situation the group adopted the following design goals:

1. Keep it simple
2. Minimize the number of intermediate representations to support "plug and play"
3. Develop redundant functionality for high-risk tools
4. Support incremental tool development and delivery
5. Support interoperability with existing software
6. Preserve and exploit type information and other program structure

2.1 Keep it simple

Although the system being designed is inherently complex, we tried to keep the design as simple as possible. Keeping things simple includes minimizing the number of interfaces for tools, using language-based file interfaces when appropriate, and using loosely coupled (separate images) instead of tightly coupled (single image) tools.

2.2 Minimize number of intermediate representations

Whenever possible, tools were designed to use the same intermediate representations. This allowed a "plug and play" configuration of the system which supported experimentation with system configuration and mitigated the risk of failure to complete individual tools.

2.3 Redundant functionality

Many specific objectives can be achieved at various stages of the pipeline. Whenever practical, critical functionality of a "high risk" tool should be duplicated someplace else in the pipeline, preferably by a lower risk tool.

2.4 Incremental development

An incremental development life cycle was adopted to allow for early integration. We chose to implement each tool in three phases, each phase resulting in a tool prototype. Prototype functionality was independently decided for each tool, but conformed to a basic structure. Prototype 1 tools included minimal functionality to support "manual" integration—that is, interfaces between tools might require some manual intervention to be operable. Prototype 2 tools included minimal functionality to support full integration and the experiment. Prototype 3 tools included additional functionality, such as performance improvements and non-critical features.

2.5 Interoperability

The generated software must interoperate with existing software. This principle strongly suggests generating source code appropriate to the target environment rather than a lower level representation.

2.6 Preserving structure

Useful structural information about a program should be preserved so the information can be used at lower stages of the pipeline. For example, potential uses of mutable state are explicit in ADL, and this information can be exploited when generating the target code. In addition, types in intermediate code at all stages of the pipeline should be consistent. In particular, the top-level functions should retain consistent types through each transformation tool so that the Ada versions of the top-level functions that are ultimately generated have appropriate types.

3 Required Functionality

The top level requirement for the system was to design a domain-specific language to describe messages in the message translation and validation (MTV) problem domain and to implement a generator that takes message specifications and produces Ada solution components. This requirement was decomposed as follows:

1. Requirements analysis. The requirements for the message specification language (MSL) and the required functionality for solution components were identified in a document [36].
2. The MSL design and compiler. The language design was a significant design deliverable [55].
3. Semantics preserving transformations on the ADL representation. The ADL language design is given in a technical report [32].
4. A translator from ADL to a Standard ML [40] based representation. In conjunction with the MSL translator (2), this translator would provide a rapid prototyping capability for the generator. Users of the generator could thus quickly compile and execute a MSL specification to test its functionality.
5. Semantics preserving transformations on the SML based representations. These transformations were required to convert higher-order programs (programs that manipulate functions as values) to first order.
6. A translator from a subset of Standard ML to Ada.

The transformations were classified as follows:

1. Program transformations to improve program efficiency
   (a) Fusion: the combination of two or more recursive function definitions (loop-like structures) into a single recursive function definition.
   (b) Deforestation: the elimination of unnecessary intermediate values in a computation.
   (c) Case smashing: a specific optimization to avoid redundant calculations in case statements and conditionals.
   (d) β-simplification: symbolically executing a program at generation time to reduce the number of run time function calls.
2. Program transformations to first-order form.
   (a) Specialization: the elimination of higher-order functions by making new instances for all function values used. Roughly speaking this is expansion of all instances of higher-order functions by "inlining".
   (b) Defunctionalization: the elimination of higher-order functions by encoding higher-order values in data structures [4]. When both methods are applicable specialization is preferred. However, defunctionalization is more general.
3. Enabling transformations

   Initially these were just the plumbing transformations and translations necessary to make certain previously developed tools fit into the pipeline. As the project developed, these became specific, potentially reusable transformations. Including:

   (a) Lambda-lifting: A transformation that moves all unnecessary anonymous functions to named top-level function definitions.
   (b) Pattern flattener: A transformation that takes an ML program that uses nested patterns and produces an equivalent program that does not.
   (c) Tree shaker: A transformation that removes unreachable definitions.
   (d) Pattern inverter: A transformation that splits SML-style function definitions into systems of equations for Astre. The transformation attempts to avoid use of conditionals and case statements, preferring to give definitions by a complete set of equations.

4 Structure of the Pipeline

The top level structure of the pipeline is shown in Figure 1. The initial set of tools proposed for the pipeline is given in Figure 2. The configuration actually used in the experiment in Phase I is shown in Figure 3. Table 1 describes the SDRR tool suite, including the tools envisioned in the initial pipeline design and those delivered with the MTV generator. Table 2 shows the correspondence of the tools in the original and delivered pipelines to the required functionality identified above.

4.1 Intermediate Representations

There are three fundamentally distinct intermediate representations visible in the pipeline: ADL, the Algebraic Design Language; SML, a general functional programming language; and RML, a restricted subset of SML.

Table 1: SDRR Tools

Tool	Description
GUI	A graphical user interface for the MSL language.
MSL Compiler	Compiles MSL message specifications to ADL programs.
HOT	Performs higher-order transformations on ADL programs.
ADL Translator	Translates ADL programs to SML.
PEP	Partial Evaluation Preprocessor. Designed to include two transformations—uncurrying and lambda-lifting.
Lambda-lifter	Converts an SML program to an RML program and performs lambda-lifting transformation on it.
Chin (Simplify)	Performs uncurrying transformation and higher-order reduction on RML programs.
Schism	A partial evaluator for Scheme programs.
Firstify	Performs defunctionalization on RML programs.
Astre	A first-order term rewriting system.
Program Instantiator	Translates RML programs to Ada.

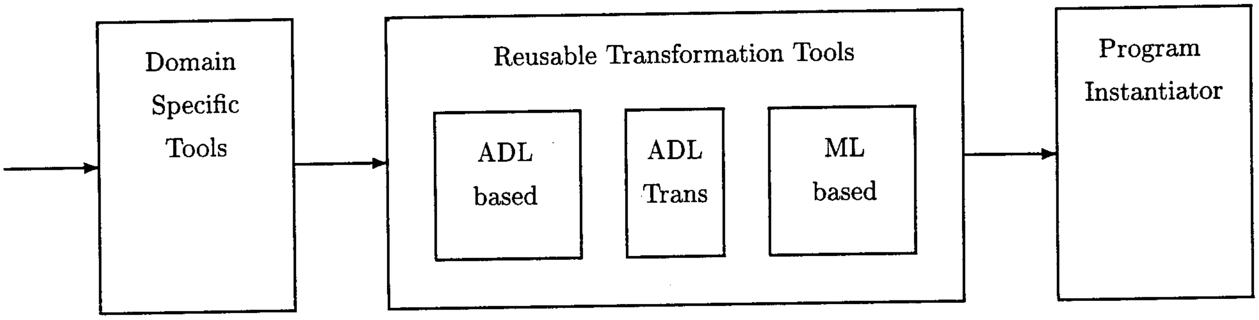


Figure 1: Top level pipeline

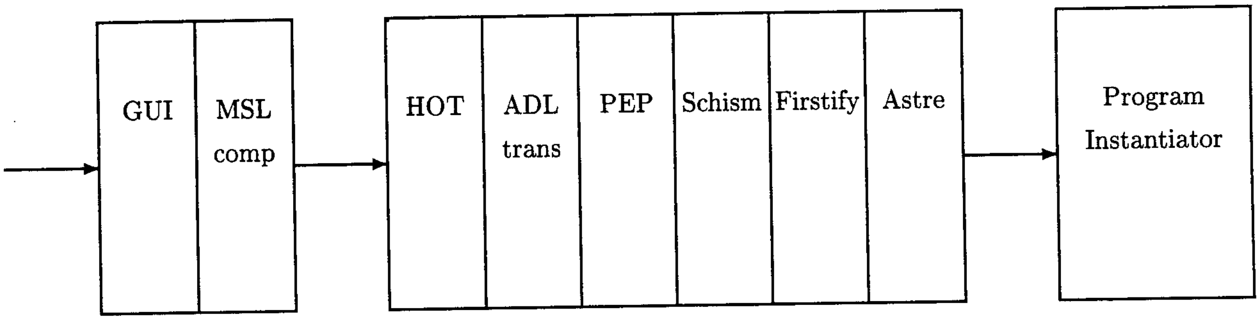


Figure 2: Original pipeline

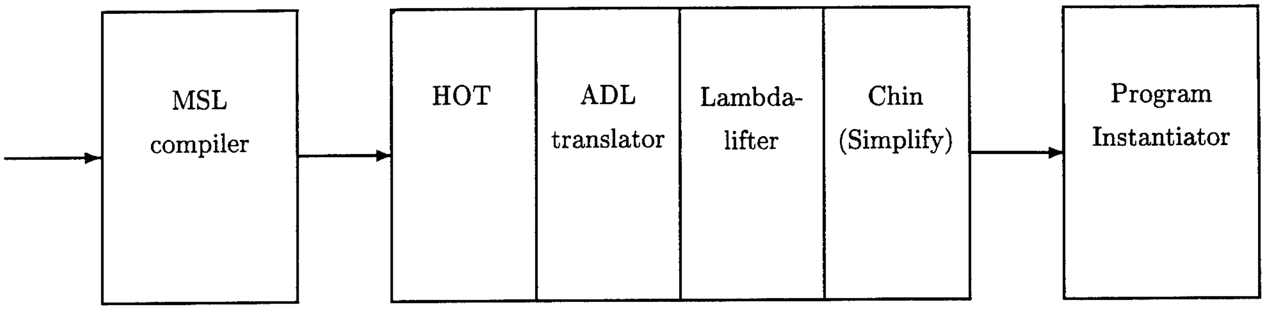


Figure 3: Delivered pipeline

Table 2: Correspondence of required functionality

Transformation	**Original**	**Delivered**
Fusion	HOT, Astre	HOT
Deforestation	HOT, Astre	HOT
Case smashing		Lambda-lifter (CRML library)
β-simplification		ADL translator
Specialization	Schism, Astre	Chin (Simplify)
Defunctionalization	Firstify	not required
Uncurry	PEP	Chin (Simplify)
Lambda-lifting	PEP	Lambda-lifter
Pattern-flattening		Lambda-lifter (CRML library)
Tree shaking		Lambda-lifter (CRML library)
Pattern inversion	ML2Astre	not required
Algebra-specific	Astre	not required

4.1.1 ADL

One of the fundamental theses of the SDRR tool suite is that generators for algorithmic problems require an intermediate form that supports extensive automatic transformation. In particular, if the domain-specific language processors are to be tractable, they must be able to produce code that is very regular, with clearly defined interfaces and invariants. These interfaces and invariants typically introduce unnecessary "administrative" calculations and intermediate values. Furthermore, the effective expression of this regular code must be flexible; at least sufficiently flexible to express, say, attribute grammar-style dependencies. It is thus necessary that the domain-specific language tools produce code in a form that can be automatically manipulated to eliminate administrative calculations and unnecessary intermediate data structures, while at the same time being sufficiently powerful to express efficient algorithms and complex abstractions.

ADL is a new language based on work in the functional programming and categorical programming paradigms [31, 32, 37]. Specifically, it builds on the calculational style of programming called *Squigol* [11, 12, 39] and the categorical programming models of Hagino [27] and of Cockett's Charity group [18, 19]. ADL extends both of its predecessors, bringing the exponential types of functional programming to the categorical languages and the rich variety of algebras in the categorical framework to the functional community.

The key properties of ADL that enable many of the transformations and analyses performed in HOT are that control is explicit and all type correct programs terminate. In particular, the fundamental control mechanisms of ADL are those for which we have the most advanced calculation techniques. ADL and HOT are the simultaneous products of an ongoing research program within PacSoft.

4.1.2 Standard ML

Standard ML is a functional programming language with imperative features. It (together with some minor variants) is the primary implementation language of the project. It was initially selected as an intermediate representation because (1) we were familiar with it, (2) we had some tools to manipulate programs expressed in it, (3) it gave us a mechanism to run our programs, which supported both rapid prototyping for generator users and incremental evaluation in tool development, and (4) it has a supported compiler with a large community of users.

In earlier work, Sheard had investigated adding reflection to Standard ML; that is, adding the ability for programs to write and evaluate fragments of programs while compiling. (This feature is found in LISP dialects.) He developed a system called Compile-Time Reflective ML (CRML), which included tools for manipulating core ML[1] [47, 45, 29, 50]. Many concepts in the SDRR pipeline were initially explored in the context of the CRML system.

At the beginning of the project it was expected that the CRML encoding of core ML would be the internal representation for all ML abstract syntax.

Furthermore, it was also expected that CRML would be used as the primary implementation language for the project because it had the capability to support the Squigol-style of programming built into ADL through its use of reflection on datatype structures [46, 28]. However, at the same time it was recognized that the restriction to core ML would prohibit us from benefiting from the rich module system provided by Standard ML; something that we expected to be a significant resource in a large system. For this reason, some tools used the full Standard ML abstract syntax.

[1] The core of ML refers to a subset that does not include the module facility. Core ML is defined in the definition of Standard ML [40].

4.1.3 RML

Restricted ML was developed in response to frustrations with the use of Standard ML as an intermediate form. Standard ML has too many features that make it convenient for programmers, but which only complicate program analysis tools. In particular, there were several recurring problems in the pipeline:

1. Standard ML patterns at least doubled the complexity of analysis routines

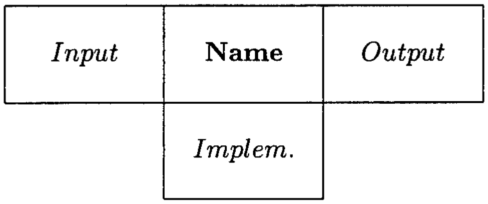
2. Standard ML's encoding of multiple argument functions as single argument functions expecting a product type did not interact well with other tools (Schism, Astre, or the program instantiator).

RML is a simplified representation of Standard ML which avoids these problems by representing only simple, non-nested patterns and providing a direct representation for multiple-argument functions. Because of the restrictions, it is necessary to perform a pattern-flattening transformation to convert Standard ML to RML.

RML was originally the internal representation of the program instantiator, but was later adopted by Lambda-lifter and Chin (Simplify) as well.

4.2 The tools in the delivered pipeline

This section summarizes the design support tools that underlie SDRR. The tools can be envisioned in terms of 'T' diagrams, such as



in which

Name is the name of the tool;

Input is the language of its input;

Output is the language of its output;

Implem. is the language in which the tool is implemented.

When the output language of tool A matches the input language of tool B, these tools can communicate directly with one another. Otherwise, a translation of representation is necessary to compose two tools.

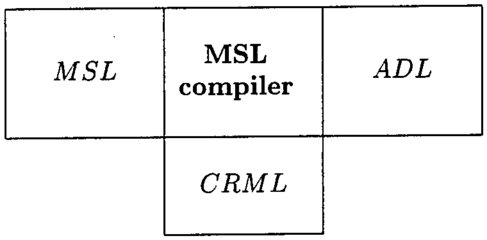
The languages indicated in this diagram are not all distinct. Table 3 summarizes the languages and their relation to one another. Of these languages, only SML and CAML have independent compilers supported by external organizations. The SML/NJ compiler is supported by AT&T Bell Laboratories and by Princeton University [1]. The CAML compiler is supported by INRIA, the French national laboratory for computer science [24]. ADL is translated directly into SML and depends upon the SML/NJ compiler, including its type checking. CRML is an extension to SML.

Table 3: Languages used in the delivered pipeline

Language	Description
MSL	Message Specification Language—a domain-specific design language
SML	Standard ML—a high-level implementation language
ADL	PacSoft's ålgebraic design language
CAML	a dialect of ML
CRML	an extension to SML
RML	a sublanguage of SML
RML_1	RML restricted to first-order functions

4.2.1 Domain-specific tools

The MSL Compiler

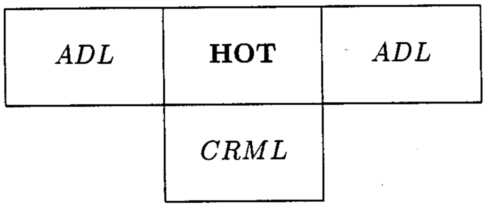


The MSL compiler translates the domain-specific Message Specification Language (MSL) to ADL. It uses traditional compiler generation technology to do lexical analysis and parsing (ML-yacc [51] and ML-lex [2], which are distributed with the SML/NJ distribution). The typechecker and code generation sections are written by hand, with some use of CRML's capability for automatic generation of control structures from datatype declarations.

The generated code is expressed in ADL, a high-level functional language. It is not optimized by the MSL compiler tool. In fact, the emphasis is on the simplicity and regularity of the code generator, so the code produced is quite naive. The code generator is intended to resemble a denotational definition of the MSL language. This style of definition exploits ADL's ability to express higher-order monadic concepts to structure the semantics. It is intended, but currently undemonstrated, that this use of monads will simplify maintenance of the compiler and support reuse for other domain-specific specification language compilers.

4.2.2 Reusable transformation tools

Higher Order Transformations



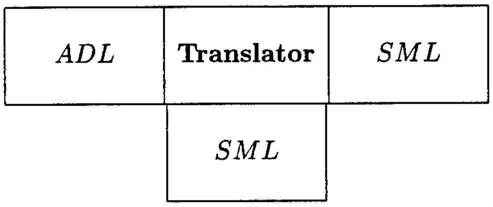
HOT unfolds and fuses definitions in ADL to improve efficiency and eliminate unnecessary intermediate structures (deforestation). In addition to the ADL program, HOT takes parameters identifying a library of known symbols. This mechanism allows the tool to treat some operations as primitives. In ADL, such collections of primitive operations acting on a type are called *concrete algebras.*

HOT uses an eager strategy to apply the transformations. This can result in code blow-up.

The transformations used in HOT to transform higher-order programs have not been previously implemented. These transformations extend the work of Malcom and others to more general datatypes and higher-order functions [38, 25, 48, 49].

The critical functionality of HOT was also present in Astre. In addition to fusion and deforestation, HOT does a significant amount of specialization. A subsequent specialization stage is still needed, however, because HOT is not guaranteed to produce first-order code.

ADL Translator



The ADL translator translates ADL into purely functional, core SML. In this translation the highly structured recursion combinators of ADL are translated into general recursive definitions in SML. The translator can be used in one of two modes. It can either be used, as it is used in the pipeline, as a tool to convert ADL programs into SML programs; or, it can be used in an interactive rapid prototype mode, where programs are translated to SML and immediately executed using the SML compiler.

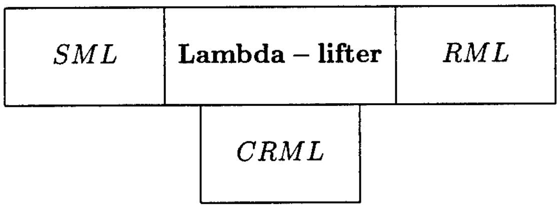
It was intended that during the course of the project, a module system would be developed for ADL. However, the proposed module system turned out to involve sufficient research questions as to be out of the scope of the current project. It was also intended that ADL have a type inference algorithm to verify the type correctness of input programs. However, since type correctness is preserved by the translation into SML, it was sufficient to "borrow" the SML type inference engine for that purpose.

The core of the ADL translator uses a straightforward translation of higher-order combinators, that while keeping the core translator simple, generates highly complex, poor quality code. The intent was that a downstream partial evaluator would be able to simplify the code. However, subsequently, the partial evaluator, Schism, was removed from the pipeline, and in system integration it was found that the quality of the code produced by ADL had a significant impact on the performance of downstream tools. Thus, a post-processor was added to the translator that performs several program simplifications using beta reduction that were able to greatly reduce the size and complexity of the code generated.

The ADL translator was originally implemented in SML. When the post-processor was added, it was written almost entirely in ADL.

ADL is a critical tool; its functionality is not duplicated anyplace else in the system.

Lambda-lifter



Standard ML programs can have embedded declarations and anonymous functions. The lambda-lifter tool lifts all such declarations to the top-level [30].

Standard ML also has a general pattern-matching facility in its function definitions and case statements. While this makes it a very appropriate programming language for writing language processing tools, it makes it inappropriate as an intermediate language in a transformation pipeline. (See Section 4.1.3 for a discussion of these issues.) Restricted ML (RML) is a restriction of ML without the pattern matching facility. In addition to lifting declarations and functions to top-level, lambda-lifter also converts programs to RML. This requires the elimination of the use of patterns (pattern-flattening).

Once patterns are flattened, it is straightforward to do an analysis of the program to determine if there are redundant computations in conditionals and cases statements. This transformation (called the case smasher) is also performed as part of the conversion to RML.

Finally, these transformations are capable of introducing unused definitions into the program. To eliminate these a transformation called the "tree shaker" is applied.

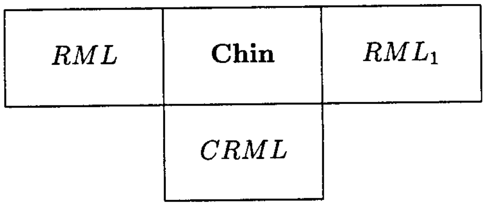
The capabilities of pattern-flattening, case-smashing, and tree shaking are all implemented in CRML and have been provided in a library because they are useful in other program transformation and analysis tools developed by PacSoft. These capabilities are tightly integrated with the lambda-lifter tool.

Lambda-lifter is a critical tool. No other tool in the pipeline translates SML to RML. None of the technology in lambda-lifter, however, was fundamentally new.

Chin (Simplify) Order-reduction consists of two parts that can be applied in sequence. The first transformation was implemented in a tool originally called *Chin*, but is now called *Simplify*. This transformation accomplishes order-reduction in almost all cases and produces a direct representation of the transformed program. But there are cases on which the algorithm used in *Chin (Simplify)* is known to be ineffective for fundamental reasons. When such cases are encountered, *Chin (Simplify)* leaves higher-order functions in the program representation.

The second transformation, *Firstify* is effective in all cases but transforms higher-order functions into data representations that may be translated into less efficient code. By applying the two algorithms in sequence, *Firstify* will affect only those rare cases on which *Chin (Simplify)* is ineffective.

In the MTV generator Chin (Simplify) works in all cases; Firstify is unnecessary.

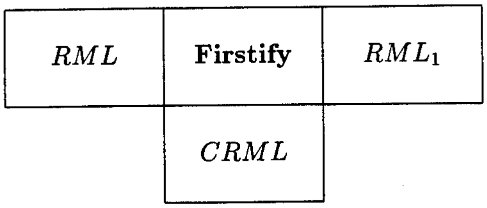


The specialization algorithm in the Chin (Simplify) tool was originally based on an unpublished paper by Chin and Darlington [17]. Chin's original algorithm had to be extended and revised to work in our environment.

In general, Chin's algorithm does not necessarily produce a first-order program. However, in the context of the MTV generator it does.

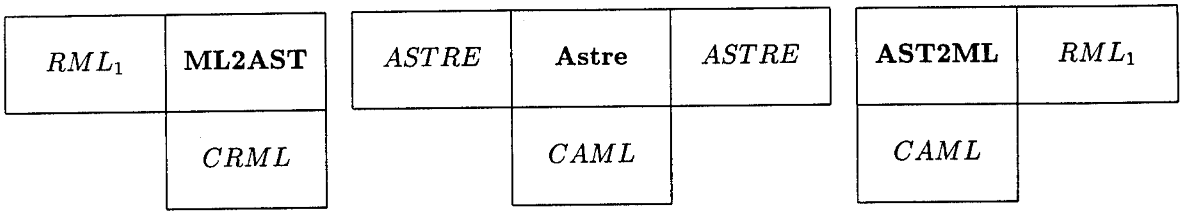
4.2.3 Reusable transformation tools not in the delivered pipeline

Firstify



Firstify was successfully developed and can be integrated into the pipeline. It is unnecessary for the MTV generator. It implements Reynolds' algorithm for defunctionalization [44, 4].

Astre



The Astre tool was successfully developed and can be integrated into the pipeline. It was not required in the MTV generator.

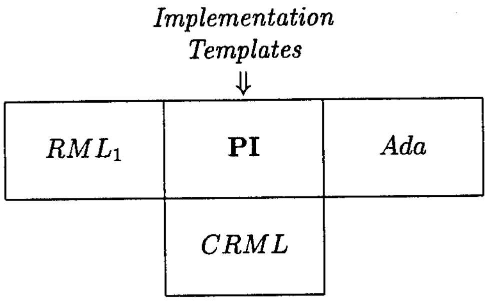
Astre is a first-order program transformation system based on term-rewriting [6, 5]. It existed as an interactive tool prior to the start of the project. Astre was extensively modified to make it function automatically.

Astre is capable of automatically performing deforestation, fusion and tupling [8, 9, 7, 10]. It can also be used interactively to accomplish a wide range of program optimizations, including the introduction of accumulator variables and other fold/unfold transformations.

The Astre system operates on sets of mutually recursive first-order functions on free algebras defined by systems of equations. A naive translation of RML programs into Astre would translate each function definition as a single equation and introduce a case statement or conditional to analyze the argument. This form of function definition is unnatural to Astre and inhibits application of most of Astre's transformations. To avoid this problem a capability called "pattern-inversion" is built into the translation tool ML2Astre.

4.2.4 Implementation Language tools

Program Instantiation



The program instantiator translates first-order RML programs into Ada packages. It is parameterized by a set of implementation templates that specify how to implement the primitive operations assumed by the RML program [42, 41].

Both RML and Ada are strongly typed. The program instantiator is distinguished because it preserves type information, naturally mapping the RML types into Ada. It does not use the traditional "boxed encoding" of values found in most functional language implementations.

Another unique feature of the program instantiator is that it translates from a parametrically polymorphic language (RML) to a language with *ad hoc* polymorphism (Ada).

Although some preliminary work had been done on the notion of program instantiation by Volpano and Kieburtz [52, 53, 54], there were no existing complete implementations.

The program instantiator is a critical tool. It has unique functionality. The concept of program instantiation is key to achieving interoperability with existing software. The concept of implementation templates is key to achieving a reconfigurable system.

The program instantiator was the largest critical tool and represented a significant risk for schedule and delivery. To mitigate this risk the implementation templates capability was given a lower priority than the type-preserving translation capability. The system configuration used in the experiment did not include the implementation templates capability.

The program instantiator was designed to be retargetable to other "target languages." In a sense, Ada is the hardest commercially-popular language to target because it is strongly typed and lacks function pointers. Retargeting to C++ or C would be a straightforward task.

4.2.5 Summary of tools

This section explains how the tools in the delivered pipeline satisfied the design goals described in Section 2.

1. **Keep it simple.** All tools have a simple, file-based interface.

2. **Minimize number of intermediate representations.** Lambda-lifter, Chin (Simplify), Firstify, and Astre all use variants of SML as their intermediate representation, thus supporting the ability to "plug and play". ADL was also used as an intermediate representation for HOT.

3. **Redundant functionality.** HOT and Astre have overlapping functionality, as do Firstify and Chin (Simplify).

4. **Incremental development.** All tools were developed incrementally; the relative priority of the functionality to be implemented in each phase of incremental development was adjusted over the course of the project.

5. **Interoperability.** The program instantiator provided interoperability with existing software.

6. **Preserving structure.** At any stage of the pipeline below the ADL translator, the types of the intermediate ML code can inferred, and the types will be consistent with any other intermediate form. The Ada code generated by the program instantiator is type-correct and consistent with the types of any other intermediate form. The issue of preserving other structural information is an open research question.

5 Design Issues Raised in Development

From the outset of the SDRR project, an evolving system design was expected and planned. The SDRR Technical Plan [15] describes the three phase incremental development process for the SDRR system, with an integrated prototype as a product of each phase. At the end of each development phase, the plan calls for a review and replan of development for the following phase. The design decisions described in this section were made as part of these planning and replanning efforts.

Other processes described in the Technical Plan, Management Plan [13], and Measurement Plan [14] were key to the success of developing software with an evolving design. The change control board consisted of the entire team, which made technical issues visible to all concerned. Weekly project meetings allowed the team to act quickly to investigate technical issues. Quarterly retreats kept the team focused on consistent goals for the pipeline. Risk assessment exercises at these retreats kept potential problems visible to tool developers. Finally, metrics data collection supported making decisions related to the development of the system.

5.1 Early design issues

5.1.1 Linear pipeline

The preliminary system design in the June 1993 planning briefing was not a strictly linear pipeline. It included a module known as tactic and control which was to select and control the transformation process, modeled after LCF-style tactics found in theorem provers and program transformation systems [26, 23]. At the time we believed that partial evaluation might be profitable at multiple stages of the pipeline, and we were concerned with maintaining "out of band" information.

The notion of tactic and control was not in the technical plan delivered in August 1993. That plan did, however, indicate some "out of band" information flow. In particular, it was anticipated that information that was known to the ADL translator would be needed by Schism, Astre and the Program Instantiator.

The information that was to be captured by this mechanism was to guide fold/unfold transformations in both Schism and Astre based on properties that were manifest in the ADL representation but no longer apparent in the ML representation. The information to be passed to the program instantiator was to help it identify where state could be introduced to improve efficiency.

The current pipeline works without the fold/unfold information because Schism and Astre are not used to perform the program transformations now achieved by HOT at the ADL level. The information to support the use of state is still potentially useful, however the program instantiator is not prepared to exploit the information at this time.

It has also proven more problematic than originally anticipated to keep this kind of information associated with program fragments in the transformation pipeline. In particular, it is hard to recognize the residual occurrences of a function (or any kind of program point) after the transformation tools have been applied.

In the delivered system the pipeline is controlled by a combination of UNIX scripts and makefiles.

Having a linear pipeline made integration and testing tractable. It contributed significantly to the simplicity, uniformity, and configurability of the system.

5.1.2 File-based interfaces

To achieve simplicity, incremental development, configurability and to simplify testing we decided at the beginning to use file-based interfaces where we explicitly parsed and unparsed the intermediate representations for at least the first two prototypes. In the end we used such interfaces exclusively.

While this decision has some performance costs, it appears to have been a very good one. On the other hand by not working to a tight integration, standards for the use of the SML module facility in tool implementations were not developed and enforced in inspections.

5.2 Design issues raised in post-prototype-one replanning

5.2.1 Schism

Schism is a partial evaluator for a subset of Scheme, an untyped LISP dialect [20, 21, 22]. Schism was developed by Charles Consel, who was initially an active participant in the project but subsequently left OGI. Ultimately, Schism was not included in the pipeline. That decision was made for two fundamental reasons: (1) technical problems related to typed and untyped languages, and (2) logistical problems because there was no resident "owner" of Schism on the team.

Partial evaluation is a transformation where a program is symbolically executed with part of its input held constant (static) and part of it unknown (dynamic). The result is a functionally equivalent program that is expected to be more efficient. A classic example is to consider a programming language with a formated print statement, such as C's printf. Typically, both the format and the values to be printed are parameters to the function. However, the format is very frequently known at compile time. In this case, it is easy to imagine the compiler interpreting the format string at compile time and producing the specialized code that prints the values. The result is a (potentially larger) program that is significantly faster because a "layer of interpretation" has been removed.

We expect naive generators to introduce several "layers of interpretation" in a typical SDRR application, so there is a great deal of appeal to partial evaluation technology.

When the project was conceived, Schism was a state-of-the-art partial evaluator and the developer was a member of our group. This was a significant incentive to attempt to bridge the gap between the typed tool suite that we were developing and the untyped partial evaluator Schism. In principle, we believed, an untyped partial evaluator would not be an insurmountable problem. It is a property of ML that if all type information is erased from an ML program and that program is symbolically executed by the equational semantics of ML then the residual program will be typeable and its type will be compatible with that of the original program.

Given this property of ML, the challenge was to develop a pair of translations between the Scheme dialect used by Schism and ML that would ensure that every Scheme evaluation of the encoded program corresponded to an ML evaluation of the original program. This technical problem was not satisfactorily solved before the logistical problems of distance became an unmanageable risk. In April, 1994, we very reluctantly decided to replace Schism with a completely new implementation of specialization for typed languages based on an algorithm by Chin.

5.2.2 Parallel development of typed specializer

During the integration of the first system prototype in the fall of 1993, problems with the integration of Schism into the pipeline (see Section 5.2.1) became apparent. Since Charles Consel had left OGI at that point, Schism was a high risk tool. For these reasons we elected to begin developing an alternative implementation of Schism's critical functionality, while still continuing to experiment with the use of Schism in the pipeline.

The alternative was an implementation of an algorithm [17] to perform specialization in a typed language. This algorithm was extensively modified to adapt it to RML, but was eventually implemented as the tool which we originally called "Chin," but we now call "Simplify." Simplify does not include all of the functionality of Schism—it just performs specialization aimed at converting a higher-order program to first order.

5.2.3 Evolution of PEP

In the initial pipeline design, a tool called Partial Evaluation Preprocessor (PEP) was to be implemented and used between the ADL translator and Schism. PEP was to have two capabilities: lambda-lifting and uncurrying. These transformations were necessary to prepare a ML program for processing by Schism and other downstream tools.

In the delivered system, lambda-lifting was implemented in a standalone tool (Lambda-lifter) which also converts SML programs to RML. A prototype uncurry transformation was implemented using the full SML abstract syntax as an intermediate form. However, this prototype was abandoned when Chin was developed, since this tool performed the uncurry transformation in addition to typed specialization.

5.3 Design issues raised in post-prototype-two replanning

5.3.1 Identification of critical and non-critical functionality

Redundant functionality of critical properties was designed into the system. In addition, tool prototypes were designed so that the third prototype included functionality that was not critical for use of the MTV generator in the Phase I experiment.

As tools proved to be problematic or unnecessary and as development resources became slim, the presence of redundant and non-critical tools made it possible to eliminate some functionality from the pipeline without impacting the experiment. Decisions concerning the elimination of non-critical functionality include the following.

1. Based on measurement data, in May of 1994 we were able to predict that we would not be able to deliver on time if we included all currently planned capabilities [34, 35]. As a result, we decided to not develop the templates capability of the program instantiator for delivery of the MTV generator.

2. As final integration and system testing of the MTV generator progressed, it became clear that the functionality of Firstify was not required. This tool was eliminated from the pipeline.

3. Françoise Bellegarde, the developer of Astre, left OGI in October 1994. Because of this, and because Astre had not yet been integrated into the pipeline, we elected to not include Astre in the delivered pipeline. The redundant functionality of Astre with HOT gave us the freedom to make this decision.

6 System Deployment

The system was deployed in a validation experiment [33]. The configuration selected for deployment was a mature version of prototype 2. Since the experiment was fundamentally testing usability and human productivity improvements with generator-based technology we did not require some of the advanced features of the third prototype. Instead we emphasized the robustness of the system in the final stages of development.

In October, 1994, PacSoft installed the "experiment baseline" version of the system at Intermetrics in Cambridge, Massachusetts, where it was used by four subjects trained in the MTV generator technology for a period of three months. They performed, collectively, a set of 12 initial message specification implementations and 56 simulated maintenance activities on these messages. The message specifications and the modifications were selected by PacSoft's contract monitors at Hanscom AFB (USAF/ESC).

In the course of the experiment, all messages were correctly specified in MSL and passed acceptance testing in the rapid prototype configuration of the system. The generator failed to produce Ada for three of the 12 initial message specifications because of capacity problems of the SDRR tools. The subjects of the experiment did not discover any other defects in the system.

The system will be further improved in an extension of the original contract, which we refer to as Phase II. The design will be carefully reviewed and more attention will be paid to performance than in Phase I, which was a proof-of-concept demonstration.

7 Conclusion

We achieved the design goals and the achievement of the goals contributed to our success.

Keep it simple While there is a significant inherent complexity in any generic generation architecture, we succeeded at keeping each tool in the system relatively small and minimized the interactions of tools to a sequential pipeline.

Minimize number of intermediate representations We restricted the intermediate representations to ADL and restrictions of Standard ML.

Redundant functionality By duplicating high-risk, critical functionality we were able to accommodate the changes inherent in any system based on unproven technology.

Incremental development By explicitly accounting for an incremental development life cycle in the management plan and the system design we were able to replan in response to schedule, personnel, and technical problems. As a result, we delivered a working system with adequate functionality on time.

Interoperability By generating source code in a language that supports a rich notion of software architecture, it is easy for generated code to interact with existing code, provided the components are described by a well defined interface specification. In the case of the MTV domain, the previous work in the domain by Plinta, Lee and Rissman made this particularly successful since it defined the interface of an MTV component as an Ada package specified in Ada PDL [43].

Preserving structure Preservation of type information gave a sanity check to all transformation tools that was useful in testing. In addition, it gave the tools rich information that enabled the reduction from higher-order to first-order to be done completely. (If the types are unknown some instances of higher-order functions cannot be identified.) Finally, it allowed us to generate a completely unboxed representation of the ML-based intermediate form in Ada.

Opportunities remain to further exploit program structure in Phase II development.

The SDRR system was developed successfully by careful planning, structured replanning and the use of a flexible design. By considering the interaction of the design and the software development life cycle we were able to build a robust system that was delivered on time and on budget.

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML compiler, August 1987. Distributed as documentation with the compiler.

[2] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML, May 1994. Distributed as documentation with ML-Lex.

[3] Jeffrey Bell et al. Software design for reliability and reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.

[4] Jeffrey M. Bell. An implementation of Reynold's defunctionalization method for a modern functional language. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, January 1994.

[5] Françoise Bellegarde. ASTRE, a transformation system using completion. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1991.

[6] Françoise Bellegarde. Program transformation and rewriting. In *Proceedings of the fourth conference on Rewriting Techniques and Applications*, volume 488 of *LNCS*, pages 226–239, Berlin, 1991. Springer-Verlag.

[7] Françoise Bellegarde. Astre: Towards a fully automated program transformation system. Technical Report 94-027, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[8] Françoise Bellegarde. Automatic transformations by rewriting techniques. Technical Report 94-009, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[9] Françoise Bellegarde. Induction and synthesis for automatic program transformation. Technical Report 94-022, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[10] Françoise Bellegarde. Termination issues in automated synthesis. Technical Report 94-028, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[11] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO Series F*. Springer-Verlag, 1986.

[12] Richard S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 52 of *NATO Series F*. Springer-Verlag, 1988.

[13] Pacific Software Research Center. Management plan for the OGI SDRR technology development and validation program, October 1993.

[14] Pacific Software Research Center. Measurement plan for the OGI SDRR technology development and validation program, October 1993.

[15] Pacific Software Research Center. Technical plan for the OGI SDRR technology development and validation program, October 1993.

[16] Pacific Software Research Center. SDRR project phase i final scientific and technical report, February 1995.

[17] Wei-Ngan Chin and John Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.

[18] J. R. B. Cockett. About Charity. Technical Report 92/480/18, University of Calgary, June 1992.

[19] J. R. B. Cockett and D. Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory, 1991*. AMS, 1992.

[20] Charles Consel. The Schism Manual, version 2.0. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1992.

[21] Charles Consel. A tour of schism. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM'93*, pages 145–154, New York, June 1993. ACM Press. 1993.

[22] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM, 1993.

[23] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1:285–326, 1985.

[24] Guy Cousineau. The categorical abstract machine. In Gerard Huet, editor, *Logical foundations of functional programming*, pages 25–45. Addison-Wesley, 1990.

[25] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *Proceedings from Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1994.

[26] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer-Verlag, Berlin, 1979.

[27] T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[28] James Hook, Richard Kieburtz, and Tim Sheard. Generating programs by reflection. Technical Report 92-015, Department of Computer Science and Engineering, Oregon Graduate Institute, July 1992.

[29] James Hook and Tim Sheard. A semantics of compile-time reflection. Technical Report 93-019, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[30] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201, pages 190–203. Springer-Verlag, 1985.

[31] Richard Kieburtz and Jeffrey R. Lewis. Programming with algebras. 1993.

[32] Richard Kieburtz and Jeffrey R. Lewis. Algebraic design language. Technical Report 94-002, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[33] Richard B. Kieburtz. Results of the sdrr validation experiment, February 1995. In [16].

[34] Alexei Kotov. Application of a new software metric in a research environment. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[35] Alexei Kotov. Measurement final report, February 1995. In [16].

[36] Jeffrey R. Lewis. A specification for an MTV generator. Technical Report 94-003, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[37] Jeffrey R. Lewis. ADL translator tool documentation, January 1995. In Collected Tool Documentation for MTV Generator (CDRL 002.7).

[38] G. Malcolm. Homomorphisms and promotability. In *Mathematics of Program Construction.* Springer-Verlag, June 1989.

[39] Lambert Meertens. Algorithmics—towards programming as a mathematical activity. In *Proc. of the CWI Symbposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[40] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[41] Dino P. Oliva. Program instantiator templates documentation, January 1995. In Collected Tool Documentation for MTV Generator (CDRL 002.7).

[42] Dino P. Oliva. Program instantiator tool documentation, January 1995. In Collected Tool Documentation for MTV Generator (CDRL 002.7).

[43] Charles Plinta, Kenneth Lee, and Michael Rissman. A model solution for C^3I message translation and validation. Technical Report CMU/SEI-89-TR-12 ESD-89-TR-20, Software Engineering Institute, Carnegie Mellon University, December 1989.

[44] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[45] Tim Sheard. Guide to using CRML. 1993.

[46] Tim Sheard. Type parametric programming. Technical Report 93-018, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[47] Tim Sheard. Type parametric programming with compile-time reflection. 1993.

[48] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.

[49] Tim Sheard and Leonidas Fegaras. Optimizing algebraic programs. Technical Report 94-004, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[50] Tim Sheard and James Hook. Meta-programming tools for ML. 1994.

[51] David R. Tarditi and Andrew W. Appel. ML-Yacc user's manual, March 1991. Distributed as documentation with ML-Yacc.

[52] Dennis Volpano. *Software Templates.* PhD thesis, Oregon Graduate Institute, October 1986.

[53] Dennis Volpano and Richard B. Kieburtz. Software templates. In *Proceedings Eighth International Conference on Software Engineering*, pages 55–60. IEEE Computer Society, August 1985.

[54] Dennis Volpano and Richard B. Kieburtz. The templates approach to software reuse. In Ted J. Biggersstaff and Alan J. Perlis, editors, *Software Reusability*, pages 247–255. ACM Press, 1989.

[55] Lisa Walton and James Hook. A preliminary definition of a domain specific design language for message translation and validation. Technical Report 94-006, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

Volume IV
Tool Survey

Tool Survey
Software Design for Reliability and Reuse Project
Phase I

Laura McKinney
Pacific Software Research Center

February 27, 1995

1 Introduction

The Pacific Software Research Center is developing a new method, *Software Design for Reliability and Reuse (SDRR)*, for the development of software component generators[1]. The generators are tailored to particular problem domains and configurable to interoperate within varied architectures and system environments. The generators function by accepting specifications written in a high-level domain-specific design language and producing software components in a wide-spectrum programming language such as Ada. Using SDRR, production of these program generators is accelerated through utilization of a suite of transformation tools which are the core of each generator. The transformation tools are reusable from domain to domain. Therefore, to complete a particular generator, a domain-specific language must be designed and a compiler developed, and the environment specifications written.

Phase I of the SDRR Project involved the creation of the reusable suite of transformation tools, and the use of these tools to develop a component generator for a particular domain. The domain selected for the project was Message Translation and Validation (MTV), a component of C^3I systems. The software environment in which instances generated by the component generator were required to operate is the Portable Reusable Integrated Software Modules (PRISM) architecture developed by the Air Force. Technical work on the project began in June 1993 and was completed in January 1995.

This document will describe the tools required to support specific aspects of the SDRR method as it was applied in the MTV domain. At the start of this project, a survey of existing tools was done and these tools were used where appropriate. Existing tools supported development in the following areas:

- System Software
- Text Processing
- Programming Languages and Systems

- Software Development Support
- Compiler Construction Tools
- System Testing
- User Interface Construction
- Program Transformation and Partial Evaluation

New tools were developed where tools did not exist or available tools did not provide adequate functionality. New or enhanced tools were developed to provide functionality in the following areas:

- Program Transformation
- Domain-Specific Design Language (DSDL) for Message Translation and Validation
- Parameterized Program Instantiation
- Test Generation
- Programming Languages and Systems

Each of the following tool descriptions will include details of function and type of use. Version and supplier information is contained in the Version Information section.

2 Existing Tools

2.1 System Software

The MTV-G was developed under SunOS. Standard Unix tools such as grep, diff, sed, and awk were used in the course of developing software. The Macintosh platform was used for support and management activities.

2.2 Text Processing

Source and data files were created using either Gnu Emacs or the Vi text editor. Documentation was created under the same tools and formatted using TeX, LaTeX and BibTeX. Microsoft Word for the Macintosh was used for some project documentation.

2.3 Programming Languages and Systems

Most tools were written in Standard ML of New Jersey (SML/NJ), a functional programming language. Several other functional languages were used for particular implementations. Tool development was done using the following languages and systems:

- Standard ML of New Jersey (SML/NJ)

- Categorical Abstract Machine Language (CAML-Light)
- SunAda
- Practical Extraction and Report Language (Perl)
- Tool command language (Tcl)/Toolkit for X-Windows (Tk)
- T (Scheme)
- Chez Scheme
- Bash shell script language

CAML-Light was used for development of Astre only. Schism was written in Scheme. Originally, T was used because it was freeware, but performance requirements motivated a move to Chez Scheme later in the project. The target language for the PRISM system was Ada, and code generated by the component generator was compiled using the SunAda compiler. A driver was developed in Ada to provide a testbed for the generated software components. Perl was used to generate scripts for analysis and collection of experiment data. Tcl/Tk were used in interface development. Bash was used to automate processes.

2.4 Software Development Process Support

Support tools were utilized for configuration management, defect tracking, metrics collection and analysis, project planning, presentation, and system test.

Configuration Management The Revision Control System (RCS) used to do configuration control for the following:

- Software artifacts
- Data and test cases
- Documentation

Customized version management scripts were developed to support use of the RCS tools in the PacSoft environment. In particular, these scripts added features for control of documents created using LaTeX, as well as adding version information explicitly to the files. Software builds were controlled using the Gnu Make utility. A combination of shell scripts and make files were used to actually run the MTV-G.

Defect Tracking The Gnats defect tracking database system was used to collect and monitor software defects.

Metrics Collection and Analysis Collection of metrics for software development was accomplished utilizing information already available under the RCS system and the Gnats defect tracking system, as well as manually collected effort data. Analysis was done using Perl scripts under Unix and Excel on the Macintosh.

Project Planning Project Scheduler 5 (PS5) for the Macintosh was used for creating Pert and Gantt charts to track progress to plan.

Presentation Presentation material was created using Microsoft PowerPoint for the Macintosh.

2.5 Compiler Construction Tools

The SML-lex scanner generator and SML-yacc parser generator were used to support the development of many transformation tools, internally-developed ML language extensions, and the DSDL for MTV.

2.6 System Test

Gnu Make was used to automate the testing process. Since the MTV-G runs as a pipeline with intermediate files as input/output between each tool, the Make facility allowed testing to commence wherever a tool was updated. This way the entire test process did not have to be rerun from the top of the pipeline, a very time-consuming activity.

2.7 User Interface Construction

A support system for domain users and demonstration purposes was developed for MTV-G using the Tcl/Tk interface development support system. This message specification development interface providing editing, compilation and testing facilities to support work using MTV-G.

2.8 Program Transformation and Partial Evaluation

Two existing program transformation tools were used during this phase of the SDRR project: Astre and Schism. The Astre [4, 5] tool is based on term-rewriting strategies to improve program efficiency, and needed extensive rework to be automated for use in the MTV-G environment. It was used successfully in early integrations of the SDRR tools, but was not included in the final MTV-G. The primary developer was no longer on-site to provide support. Schism [7] is a partial evaluator written to operate on programs written in Scheme. To integrate Schism into MTV-G, input and output translators were required. Other significant technical barriers existed to the complete integration of Schism, and due to the timeliness of potential availability of the tool in a usable state, it was dropped from use midway through the project.

3 New Tools

Since this was a research project, many new tools were developed to support new technology. The core program transformation technology was developed with little reliance on existing transformation tools.

3.1 Program Transformation

Program transformation was a major area of research during this project. A particular focus was the design and development of Algebraic Design Language (ADL) [8] which expresses the semantics of specifications written in domain-specific languages. ADL provides the structure necessary for program transformation that is unavailable in any other existing language. A translator from programs expressed in ADL to programs in SML was written to provide an executable platform for ADL.

A new program transformation tool, Higher-Order Transformation (HOT) [10], was developed to operate on programs written in ADL and to exploit the structure expressed in those programs to perform meaning-preserving transformations to improve program efficiency.

Once a program has been transformed by HOT and translated to SML by the ADL translator, it is ready for preparation for translation into a wide-spectrum language. For an SML program to be translated into Ada, some order reduction must be done. Several tools were developed to perform aspects of this transformation: Chin (Simplify) [6] (which uses Chin's algorithm to make programs first order), Firstify [2] (which also reduces programs to first order), and Lambda Lifter [3] (which "lifts" all declarations to the top level).

The other existing transformation tools operated on programs expressed in Scheme (Schism) or in a term system (Astre). Translations from SML to and from these formats were written to allow these tools to operate in the tool suite.

3.2 Domain-Specific Design Language for MTV

The creation of a domain-specific design language and an attendant compiler is a key step in the use of the SDRR method. The compiler must emit code in ADL. For the MTV domain, there were no existing tools that provided a high-level, specification language for message translation and validation with semantics expressed in ADL.

An existing domain solution, the Model Solution MTV [9], provided a suite of reusable Ada templates for development of MTV modules. No aspects of this solution were considered reusable as part of the development of the DSDL for MTV, other than the thorough domain analysis which had been done for the requirements phase of development of the Model Solution. The implementation of the Model Solution also provided a model for the integration requirements of the PRISM system. Instances generated by the new program generator were required to interoperate within PRISM in the same manner as Model Solution modules.

Message Specification Language (MSL) Compiler A new tool, the Message Specification Language (MSL) [11] compiler, was developed to compile specifications written in a declarative specification language expressive over the MTV domain. The MSL compiler developed for this project produced a set of translation functions expressed in ADL and providing equivalent functionality to MTV programs developed using the Model Solution. These translation functions provide the necessary interface to the PRISM system.

User Interface for MSL A support system for domain users and demonstration purposes was developed for MTV-G using Tcl/Tk interface development support system. This message specification development interface providing editing, compilation and testing facilities supported work in MSL.

3.3 Program Instantiation

The last step is a translation from SML to the target language for the environment within which instances generated must operate. For PRISM, this is Ada. A Program Instantiator from SML to Ada was written to perform this translation. Integration and interoperability parameters expressed using environment specifications controlled this translation and allowed the generated components to interoperate within PRISM. A source code Splitter tool was written to overcome SunAda compiler limitations by breaking large Ada source files into sets of smaller source code files.

3.4 Test Generation

An automated test case generator was developed to emit both positive and negative test case messages for any given MSL specification. The tool development relied on reuse of portions of the MSL compiler.

3.5 Programming Languages and Systems

An extension to basic SML was developed to provide compile-time reflection (Compile-time Reflective ML — CRML). This research feature reduced software development effort while providing additional ancillary research opportunities in ML.

4 Version Information

Tool	Version	System	Vendor
SunOS	v. 4.1.3	Unix	Sun Microsystems, Inc.
TeX, LaTeX and BibTeX	v. 3.1415, C v. 6.1	Unix	Knuth/Lamport
MS Word	v. 5.1, 6.0	Macintosh	Microsoft
Gnats	v. 3.2	Unix	Free Software Foundation
Excel	v. 4.0, 5.0	Macintosh	Microsoft
Project Scheduler 5	v. 1.0	Macintosh	Scitor Corporation
PowerPoint	v. 3.0, 4.0	Macintosh	Microsoft
SunAda Compiler	v. 1.1(j)	Unix	Sun Microsystems
SML/NJ Compiler	v. 0.93	Unix	AT&T
CAML-Light Compiler	v. 3.1	Unix	INRIA
Perl	v. 4, patch level 36	Unix	Larry Wall
SML Lexgen	v. 1.4	Unix, SML/NJ	AT&T
SMLyacc	1989	Unix, SML/NJ	AT&T
Gnu Make	v. 3.6	Unix	Free Software Foundation
Tcl/Tk	v. 7.3/3.6	Unix	UC Berkeley
T (Scheme)	v. 3.1	Unix	Yale University
Chez Scheme	v. 4.1q	Unix	Cadence Research Systems
RCS	v. 5.6	Unix	standard Unix distribution
Bash	v. 1.14.2	Unix	Free Software Foundation

5 Conclusion

Whenever possible, existing tools were used in development of the MTV-G. Since the entire program transformation suite of tools, the domain-specific language and the program instantiator were developed to satisfy research needs and prove a new technology, extensive reuse within these areas was not possible. In other areas such as system support and project support, existing technology was leveraged appropriately.

References

[1] Jeffrey Bell et al. Software design for reliability and reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.

[2] Jeffrey M. Bell. An implementation of Reynold's defunctionalization method for a modern functional language. Master's thesis, Oregon Graduate Institute, January 1994.

[3] Jeffrey M. Bell. Lambda-lifter tool documentation, January 1995. In Collected Tool Documentation for MTV Generator (CDRL 002.7).

[4] Françoise Bellegarde. Astre: A transformation system for first-order functional programs, November 1994.

[5] Françoise Bellegarde. Astre: Towards a fully automated program transformation system. Technical Report 94-027, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[6] Wei-Ngan Chin and John Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.

[7] Charles Consel. New insights into partial evaluation: the Schism experiment. In *ESOP'88*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246. Springer-Verlag, March 1988.

[8] Richard Kieburtz and Jeffrey R. Lewis. Algebraic design language. Technical Report 94-002, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[9] Charles Plinta, Kenneth Lee, and Michael Rissman. A model solution for C^3I message translation and validation. Technical Report CMU/SEI-89-TR-12 ESD-89-TR-20, Software Engineering Institute, Carnegie Mellon University, December 1989.

[10] Tim Sheard. HOT tool documentation, January 1995. In Collected Tool Documentation for MTV Generator (CDRL 002.7).

[11] Lisa Walton and James Hook. Message specification language (MSL): A domain specific design language for message translation and validation. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, January 1995.

Volume V
Algebraic Design Language (Definition)

An Algebraic Design Language (Definition)

Richard B. Kieburtz and Jeffrey Lewis
Pacific Software Research Center
Oregon Graduate Institute
of Science & Technology
P.O. Box 91000
Portland, OR 97291-1000 USA

February 28, 1995

Abstract

This report constitutes a definition of a new, high-level programming language called ADL. It uses the mathematical concept of structure algebras as the basis for defining computations. When algebras are used to specify programs, control structure is fixed first and data structure, or representations, second. There is no explicit recursion or iteration construct in ADL. Control is determined by combinators applied to inductively defined algebras. An intended use of ADL is to provide computational semantics of specialized software design languages.

ADL also makes use of coalgebras, a concept dual to that of algebras. With coalgebras, iterative control structures typical of search algorithms can be specified.

There is a strong notion of type in ADL, guaranteeing that all well-typed programs terminate. This allows us to use sets as ADL's semantic domain and to provide ADL with an equational logic. However, to check the type correctness of an expression, there can be proof obligations that cannot be discharged mechanically. A benefit of the equational logic is that an ADL program is amenable to transformation based upon the equational theories of its algebras. Transformations are not discussed in this report, however.

1 Introduction

ADL—Algebraic Design Language—is a higher-order software specification language in which control is expressed through a family of type-parametric combinators, rather than through explicitly recursive function definitions. ADL is based upon the mathematical concept of structure algebras and coalgebras. The declaration of an algebraic signature specifies a variety of structure algebras[1]. A signature declaration implicitly defines the terms of one particular algebra, the free term algebra of the signature, which corresponds to a datatype in a typed, functional programming language such as ML, Haskell or Miranda.

Classes of coalgebras are declared by record signatures. Elements of free coalgebras correspond to infinite records, and have no direct analogy in most conventional programming languages, although streams, which can be created in lazy functional languages, provide one such instance.

The functions definable in ADL are the λ-definable morphisms of structure algebras and coalgebras. Properties of such functions can be proved by applying rules of inductive (or co-inductive) inference dictated by the structure of the underlying signature.

There are related studies of the use of higher-order combinators for theoretical programming [MFP91, Fok92], however, none has yet been incorporated into a practical system for program development. The origin of such techniques appears to lie in the work of the *Squiggol* school [Bir86, Bir88, Mee86], subsequently influenced by a thesis by Hagino [Hag87] in which datatype morphisms are generalized in a categorical framework. A categorical programming language called *Charity* [CS92] embodies inductive and coinductive control structures based upon a categorical framework. The characterization of datatypes as structure algebras (and coalgebras) [Mac71] can be attributed to Hagino.

ADL has syntax similar to that of the ML language family. Unlike Standard ML, ADL has no **ref** types and has no **rec** definitions. ADL can be given a simple semantics over sets.

[1] A *variety* is a class of algebras that have a common signature.

However, the domain of a function may be subject to a logically formulated restriction, and complete type-checking of ADL programs is only semi-decidable. The semantics of ADL does not rely upon fixed points of functionals and does not require domains of CPO's as an underlying structure, although such an interpretation is certainly possible.

2 Algebras, Types and signatures

ADL is a higher-order, typed language whose type system is inspired by concepts from the theory of order-sorted algebras, from Martin-Löf's type theory and from the Girard-Reynolds second-order lambda calculus. While ADL does not provide the full generality of the second-order lambda calculus, it does distinguish between the names of types and the semantics of types and it contains combinators that are indexed by the names of type constructors. Its type system is sufficiently rich that type-checking is not known to be decidable.

Nevertheless, the ADL type system is amenable to an abstract interpretation that is similar to the Hindley-Milner system with consistent extensions. Type inference in the Hindley-Milner system, while of exponential complexity in the worst case, has been shown to be feasible in practice through years of experience with its use in the ML family of languages. The Hindley-Milner system, which embodies a structural notion of type, guarantees the slogan

"Well-typed programs don't go wrong".

This means that programs which satisfy the structural typing rules respect the signatures of multi-sorted algebras—integer data are never used as reals or as functions, for instance. ADL adds to the structural typing restrictions the further requirement that

"Well-typed programs always terminate".

This implies that the type system accommodates the precise description of sets that constitute the domains of functions definable in ADL. Accurate type-checking in ADL requires the construction of proofs of propositions. This task is made substantially easier than it would be in an untyped linguistic framework by the underlying approximation furnished by structural typings.

In Standard ML and related languages, the Hindley-Milner type system is extended with datatype declarations. A datatype declaration names a type and specifies a finite set of data constructors. A datatype name may have one or more type variables as parameters, and thus actually names a type constructor. When a type variable is introduced as a parameter in a datatype declaration, the variable is bound by abstraction, rather than universally quantified. The binding occurrence of an abstracted type variable is its occurrence in the left hand side of a datatype definition. The scope of a type variable extends over the right hand side of the datatype definition. Application of a type constructor to a type expression can be understood syntactically, as the substitution of the argument expression for all occurrences of the type variable in the datatype declaration.

In ADL, datatype declarations are generalized to signature declarations that specify algebraic varieties. Following the conventions of multi-sorted algebras, we call the names of types and type constructors *sorts.* The generalization can be summarized in the following table:

Parameterized datatypes	**Varieties**
type	algebra
type constructor	sort
data constructor	operator

The *arity* is a syntactic property of a sort. The arity indicates how to form type expressions from sorts. A sort with nullary arity, designated by $*$, is said to be *saturated.* A sort with non-nullary arity, designated by $* \rightarrow *$, $(*, *) \rightarrow *$, $(*, *, *) \rightarrow *, \ldots$ is said to be *unsaturated.* An unsaturated sort, s, can form a saturated sort expression by applying it to a tuple that consists of as many saturated sort expressions as there are asterisks to the left of the arrow in the arity of s. A type name in ADL is a saturated sort expression. Type variables range over saturated sort expressions.

ADL departs significantly from functional programming languages such as SML by providing declarations of signatures that define classes of structure algebras, not simply datatypes. An algebraic signature consists of a finite set of operator names, together with the type of the domain of each operator. The codomain of an operator is the carrier type for each particular algebra.

2.1 Some familiar algebras

Where we would write the declaration of a *list* datatype in Standard ML as

```
datatype 'a list = nil | cons of ('a *  'a list)
```

a corresponding declaration of a family of *List* signature-algebras is written in ADL as:

```
signature List(a){type c; list/c = {$nil, $cons of (a * c)}}
```

A signature algebra is characterized by a carrier and a set of typed operators. A variety of signature algebras specifies neither the carrier nor the operators, but only their typings, in terms of the carrier, the type parameters of the variety. and possibly some constant types.

The above declaration asserts *List*(a) to be the name of a variety of algebras that is parameterized upon a single type (designated by the type variable a) and further declares that the variety has a single sort, *list*. The sort is unsaturated because the variety has a type parameter; thus *list* $: * \rightarrow *$. The type variable c stands for the carrier type. The signature declares that there are two operator symbols of sort *list*, with typing:

$$\begin{array}{ll} \text{type } a,\, c \ \vdash & \$Nil : c \\ & \$Cons : a \times c \rightarrow c \end{array}$$

Properly, the operator *$Nil* could have been given a function type, $\mathbf{1} \rightarrow c$, by declaring it as "*$Nil* **of 1**". Since **1** is a singleton set, every function in the type $\mathbf{1} \rightarrow c$ is isomorphic to an element in c.

Operator names always begin with '$' to distinguish them from other identifiers. A concrete algebra is specified by a structure that contains bindings for the carrier type and for each operator of the algebra.

Each signature declaration implicitly defines one specific datatype. This is the type of free terms, whose operators are the free constructors of the signature (just as for SML datatypes) and whose elements are the terms constructed by well-typed applications of these constructors. The names of the data constructors of the datatype of free terms are derived from the names of operators in the signature by dropping the initial '$' symbol.

The signature algebra whose operators are data constructors is an important special case. Data constructors are unconstrained by equational laws. The set of terms generated from values of a type a by well-typed applications of the data constructors of an unsaturated sort t gotten from a signature T constitutes a datatype that we call $t(a)$. Under suitable constraints on a signature T, the algebra generated by its data constructors is unique modulo the isomorphism class of the parameter, a. This is called the *free term algebra* of the variety $T(a)$.

2.2 Structure algebras

There is another notion of algebras that we find useful. The concept of A structure algebra is independent from that signature algebras. It has its roots in category theory. However, the particular form of structure algebras in which we are interested is related to signature algebras, and this can cause confusion if not understood. A structure algebra can be characterized by a free signature algebra with a type parameter, plus an operator whose domain is the carrier of the free algebra. The codomain of this operator is coincident with the parameter type of the free algebra. This characterization is summarized in the following definition:

Definition 2.1 : Let T be a parameterized signature with a single sort, $t : * \rightarrow *$. A *T-structure algebra* (or T-algebra, for short) is a pair (c, h), where c is a type called the *carrier* of the algebra and $h : t(c) \rightarrow c$ is called its structure function.

□

A *T-algebra morphism* is a function that maps one T-algebra to another. This notion can be made precise, but we need some notation to express it.

Definition 2.2 (Notation) : A signature declaration lists the operators of a sort s as

$$\kappa_1 \textbf{ of } \sigma_1, \ldots, \kappa_n \textbf{ of } \sigma_n$$

in which the κ_i stand for the operators and the σ_i are saturated sort expressions, each of the form of one or more factors, i.e. $\sigma_i = \sigma_{i,1} * \ldots * \sigma_{i,m_i}$.

□

The meaning of T as a parameterized signature can be extended to define a signature as the mapping of a function. The following definitions have been specialized to the case of a single-sorted signature.

Definition 2.3 : Let $T(a)$ be a signature with a single sort, t, and let $f : a \rightarrow b$. Let $\kappa_1, \ldots, \kappa_n$ be the operator
names of the signature T. then $map\_t\, f : t(a) \rightarrow t(b)$ is defined as follows:

$$map\_t\, f = \lambda x.\ \textbf{case } x \textbf{ of}$$

$$\vdots$$

$$\kappa_i(x_1, \ldots, x_{m_i}) \Rightarrow \kappa_i(y_1, \ldots, y_{m_i})$$

$$\text{where } y_j = \begin{cases} f\, x_j & \text{if } x_j : a \\ map\_t\, f\, x_j & \text{if } x_j : t(a) \\ map\_s\,(map\_t\, f)\, x_j & \text{if } x_j : s(t(a)) \text{ where } s \text{ is an unsaturated} \\ & \qquad\qquad\quad \text{sort of a signature } T' \\ x_j & \text{if } x_j : s' \text{ where } s' \text{ is a saturated sort} \end{cases}$$

□

Definition 2.4 : Let $T(a)$ be a signature with a single sort, t. A *T-algebra morphism* is a function $f : a \rightarrow b$ that satisfies the commuting diagram below:

$$\begin{tikzcd}
t(a) \arrow[r, "map\_t\, f"] \arrow[d, "h"'] & t(b) \arrow[d, "k"] \\
a \arrow[r, "f"] & b
\end{tikzcd}$$

The diagram displays the equation

$$f \circ h = k \circ map\_t\, f$$

Note that both h and k are structure functions.

□

Example 2.1 : A *list*-algebra morphism. Let $exp2 = \lambda n.\, 2^n$, and let *sum* and *product* be the functions that reduce a list of non-negative integers by addition and multiplication, respectively. Then the following diagram illustrates *exp2* as a morphism of *list*-algebras:

$$\begin{tikzcd}
\mathit{list}(\mathit{int}) \arrow[r, "\mathit{map\_list}\ \mathit{exp2}"] \arrow[d, "\mathit{sum}"'] & \mathit{list}(\mathit{int}) \arrow[d, "\mathit{product}"] \\
\mathit{int} \arrow[r, "\mathit{exp2}"'] & \mathit{int}
\end{tikzcd}$$

□

When a signature declaration satisfies the unitary condition (see Definition 2.7), there exists for each type a, a free term algebra, $(t(a), \mu_a)$. (This condition is sufficient but not necessary.) It has the property that for any T-algebra (a, h), the homomorphism h is the unique T-algebra morphism from the initial term algebra to (a, h). The last statement is summarized in the following commuting square:

$$\begin{tikzcd}
t^2(a) \arrow[r, dotted, "\mathit{map\_t}\,h"] \arrow[d, "\mu_a"'] & t(a) \arrow[d, "h"] \\
t(a) \arrow[r, dotted, "h"] & a
\end{tikzcd}$$

The dotted arrow indicates that the function that makes the diagram commute is uniquely determined from the other data in the diagram.

Definition 2.5 Let $t : * \rightarrow *$ be an unsaturated sort of a signature T. An element "κ **of** σ" of the $t(a)$ component of T is called a *unit operator* if $\sigma = \sigma_1 * \cdots * \sigma_m$ and there is at least one occurrence of a among the σ_i. If $\sigma = a$, then κ is said to be *perfect.*

Definition 2.6 A signature T is *zero-based* if it contains a unique element "κ_0 **of 1**".

Definition 2.7 A singly parameterized signature T with a single sort, $t : * \rightarrow *$, is *unitary* if it contains a unique unit operator, and either

1. the unit operator is perfect, or

2. if the unit operator is given by a signature component "κ of $\sigma_1 * \cdots * \sigma_m$" then

- only one factor, σ_i, is a, the type parameter of the signature,
- for all factors, $\sigma_j \neq a \Rightarrow \sigma_j = c$, where c is the type variable declared to represent the carrier of type t(a),
- T is zero-based.

□

If a signature T is unitary, the datatype of free terms of T is the carrier of an algebra. This algebra is in fact, initial in the category of T-algebras. Thus a (redundant) ADL specification of the free term algebra for the sort *list* would be

```
List(a){c := list(a); list{$Nil := Nil, $Cons := Cons}}
```

It is important to remember the distinction between data constructors in the free term algebra and operators in the signature of an algebra. Different instances of an operator may have different types, depending upon the environment in which the operator appears, as the carrier type will differ in distinct algebras of the same family. The data constructors are a special case of the operators for one specific algebra, and their types are fixed, up to variation in the type argument of an unsaturated sort.

Example 2.2 : Three different *List*-algebras are:

```
List(int){c := int; list{$Nil := 0, $Cons := (+)}}
List(a){c := int; list{$Nil := 0, $Cons := \(x,y) 1+y}}
List(a){c := list(a) -> list(a);
        list{$Nil := id,
             $Cons := \(x,f) \y Cons(x,f y)}}
```

where *id* is the polymorphic identity function, here instantiated with the type *list*(a). These *List*-algebras induce homomorphisms from free *List*-algebras that represent functions that sum a list of integers, calculate the length of a list, and catenate two lists, respectively. A combinator to specify these homomorphisms will be introduced in the next section.

When a signature in ADL has only a single sort, as does *List*, an algebra specification may be abbreviated by omitting the inner set of curly braces and the sort name that is prefixed to the opening brace. Thus we could abbreviate the first algebra in the list of examples above, as

```
List{c := int; $Nil := 0, $Cons := (+)}
```

Here are the declarations of some other signatures that define useful varieties of algebras in ADL:

```
signature Nat{type c; nat/c = {$Zero, $Succ of c}}
signature Tree(a){type c; tree/c = {$Tip of a, $Fork of (c * c)}}
signature Bush(a){type c; bush/c = {$Leaf of a, $Branch of list(c)}}
```

Note that *nat* is a saturated sort, while *tree* and *bush* are both unsaturated.

2.3 The reduce combinator

If t is an unsaturated sort of the signature T, a structure function of the class of T-algebras is any function $h : t(a) \to a$. If T has a free term algebra, then h is also the unique T-algebra morphism from $(t(a), \mu_a)$ to (a, h), and we call it a *homomorphism.* (Recall that the meaning of "morphism" is "form-preserving". Here the form that is preserved is the underlying structure of the algebra.) More generally, the composition of a T-algebra morphism $f : a \to b$ with a homomorphism, i.e. $g = f \circ h : t(a) \to b$, is a T-algebra morphism from the free term algebra, and is uniquely determined by the algebra of its codomain.

ADL defines a combinator, *red* , that takes an algebra specification to a free-algebra morphism. The *red* combinator obeys a homomorphism condition for each algebra on which it is instantiated. For the algebras we have considered, these equations are:

$$\begin{aligned}
\mathit{red}[\mathit{nat}]\ \mathit{Nat}\{c; \$\mathit{Zero}, \$\mathit{Succ}\}\ \mathit{Zero} &= \$\mathit{Zero} \\
\mathit{red}[\mathit{nat}]\ \mathit{Nat}\{c; \$\mathit{Zero}, \$\mathit{Succ}\}\ (\mathit{Succ}\, n) &= \$\mathit{Succ}\,(\mathit{red}[\mathit{nat}]\ \mathit{Nat}\{c; \$\mathit{Zero}, \$\mathit{Succ}\}\ n) \\
\mathit{red}[\mathit{list}]\ \mathit{List}\{c; \$\mathit{Nil}, \$\mathit{Cons}\}\ \mathit{Nil} &= \$\mathit{Nil} \\
\mathit{red}[\mathit{list}] : \mathit{List}\{c; \$\mathit{Nil}, \$\mathit{Cons}\}\ (\mathit{Cons}\,(x, y)) &= \$\mathit{Cons}\,(x,\ \mathit{red}[\mathit{list}]\ \mathit{List}\{c; \$\mathit{Nil}, \$\mathit{Cons}\}\ y\ )
\end{aligned}$$

$$
\begin{aligned}
\mathit{red}[\mathit{tree}]\ \mathit{Tree}\{c; \$\mathit{Tip}, \$\mathit{Fork}\}\ (\mathit{Tip}\,x) &= \$\mathit{Tip}\,x \\
\mathit{red}[\mathit{tree}]\ \mathit{Tree}\{c; \$\mathit{Tip}, \$\mathit{Fork}\}\ (\mathit{Fork}(l, r)) &= \$\mathit{Fork}(\mathit{red}[\mathit{tree}]\ \mathit{Tree}\{c; \$\mathit{Tip}, \$\mathit{Fork}\}\ l, \\
&\qquad\qquad \mathit{red}[\mathit{tree}]\ \mathit{Tree}\{c; \$\mathit{Tip}, \$\mathit{Fork}\}\ r) \\
\mathit{red}[\mathit{bush}]\ \mathit{Bush}\{c; \$\mathit{Leaf}, \$\mathit{Branch}\}\ (\mathit{Leaf}\,x) &= \$\mathit{Leaf}\,x \\
\mathit{red}[\mathit{bush}]\ \mathit{Bush}\{c; \$\mathit{Leaf}, \$\mathit{Branch}\}\ (\mathit{Branch}\,y) &= \$\mathit{Branch}(\mathit{map\_list}(\mathit{red}[\mathit{bush}]\ \mathit{Bush}\{c; \$\mathit{Leaf}, \$\mathit{Branch}\})\ y)
\end{aligned}
$$

The function *map_list*, referred to in the last equation above, will be defined below.

2.3.1 Some *List*-algebra homomorphisms

Here are some examples of *List*-algebra morphisms constructed with *red* [*list*] and the algebra specifications given in Example 2.2:

```
sum_list = red[list] List(int){c := int; $Nil := 0, $Cons := (+)}
length = red[list] List(a){c := int; $Nil := 0, $Cons := \(x,y) 1+y}
append = red[list] List(a){c := list(a) -> list(a);
                           $Nil := id,
                           $Cons := \(x,f) \y Cons(x, f y)}
```

Further examples of *List*-algebra morphisms are:

```
map_list f = red[list] List(a){c := list(b);
                               $Nil := Nil,
                               $Cons := \(x,y) Cons(f x, y)}
```

where `f` has the type `a -> b`, and

```
flatten_list = red[list] List(list(a)){c := list(a); $Nil := Nil, $Cons := append}
```

The typings of the constants defined by these equations are:

```
sum_list : list(int) -> int
length : list(a) -> int
append : list(a) -> list(a) -> list(a)
map_list : (a -> b) -> list(a) -> list(b)
flatten_list : list(list(a)) -> list(a)
```

2.3.2 Further examples

Some examples of *nat*-algebra morphisms are:

```
ntoi = red[nat] Nat{c := int; $Zero := 0, $Succ := \n 1+n}

add x = red[nat] Nat{c := nat; $Zero := x, $Succ := Succ}

plus = red[nat] Nat{c := int -> int; $Zero := id, $Succ := \f \n 1 + f n}
```

with typings

```
ntoi : nat -> int

add : nat -> nat -> nat

plus : nat -> int -> int
```

Examples of *tree* morphisms are:

```
sum_tree = red[tree] Tree(int){c := int;
                               $Tip := id,
                               $Fork := (+)}
list_tree = red[tree] Tree(a){c := list(a);
                              $Tip := \x Cons(x,Nil),
                              $Fork := \(x,y) append x y}
map_tree f = red[tree] Tree(a){c := tree(b);
                               $Tip := \x Tip(f x),
                               $Fork := Fork}
flatten_tree = red[tree] Tree(tree(a)){c := tree(a);
                                       $Tip := id,
                                       $Fork := Fork}
```

with typings

```
sum_tree : tree(int) -> int

list_tree : tree(a) -> list(a)
```

```
map_tree : (a -> b) -> tree(a) -> tree(b)

flatten_tree : tree(tree(a)) -> tree(a)
```

The analogous examples of *bush* morphisms are:

```
sum_bush = red[bush] Bush(int){c := int;
                               $Leaf := id,
                               $Branch := sum_list}
list_bush = red[bush] Bush(a){c := list(a);
                              $Leaf := \x Cons(x,Nil),
                              $Branch := flatten_list}

map_bush f = red[bush] Bush(a){c := bush(b);
                               $Leaf := \x Leaf(f x),
                               $Branch := Branch}
flatten_bush = red[bush] Bush(bush(a)){c := bush(a);
                                       $Leaf := id,
                                       $Branch := Branch}
```

with typings

```
sum_bush : bush(int) -> int
list_bush : bush(a) -> list(a)
map_bush : (a -> b) -> bush(a) -> bush(b)
flatten_bush : bush(bush(a)) -> bush(a)
```

Exercise 2.1 Reverse of a list

a. Specify a *list*-reduce to compute the reverse of a list.

b. Now specify a second *list* reduce with carrier type $list(a) \rightarrow list(a)$ to define a function $rev : list(a) \rightarrow list(a) \rightarrow list(a)$ that satisfies the equation

$$rev\, x\; Nil = reverse\, x$$

2.4 Primitive recursion and the case analysis combinator

Recall Kleene's primitive recursion scheme to define functions on natural numbers:

$$\begin{aligned} f\,(\mathit{Zero}, x_1, \ldots, x_n) &= g\,(x_1, \ldots, x_n) \\ f\,((\mathit{Succ}\,n), x_1, \ldots, x_n) &= h\,(\mathit{Succ}\,n,\ f\,(n, x_1, \ldots, x_n),\ x_1, \ldots, x_n) \end{aligned}$$

where $g : t_1 \times \ldots \times t_n \rightarrow a$ and $h : \mathit{nat} \times a \times t_1 \times \ldots \times t_n \rightarrow a$. Although the primitive recursion scheme can be represented as a *nat*-reduce, the representation is unnatural and if implemented directly, can result in algorithms with worse-than-expected performance. For instance, the **case** expression for type *nat* when expressed as a *nat*-reduce is

$$\begin{array}{lcl} \textbf{case } x \textbf{ of} & = & \mathit{snd}(\mathit{red}[\mathit{nat}]\ \mathit{Nat}\{c := \mathit{nat} \times a; \\ \quad \mathit{Zero} \Rightarrow g & & \qquad\qquad\qquad\qquad \$\mathit{Zero} := (\mathit{Zero},\ g), \\ \mid\ \ \mathit{Succ}(x') \Rightarrow h\ x' & & \qquad\qquad\qquad\qquad \$\mathit{Succ} := \lambda(x, y)\ (\mathit{Succ}\,x,\ h\,x)\}) \\ \textbf{end} & & \end{array}$$

Evaluation of the *nat*-reduce explicitly traverses the entire structure of a term to construct the argument needed in the successor instance of the case analysis. This takes time linear in the size of a *nat* term, whereas the *case* primitive is a constant time function.

To avoid the problem of introducing a computation time penalty for a primitive control structure, **case** has been adopted as a control combinator in ADL, with syntax similar to that of Standard ML for the benefit of familiarity. The domain of a **case** combinator is the free datatype generated by an algebraic variety.

A primitive recursive function is, however, a structure function of *nat*-algebras, one in which the carrier always has the form $\mathit{nat} * a$ for some type a. A function definable by primitive recursion can be calculated by taking the second projection of a pair calculated by a *Nat*-reduce. For example, a factorial function can be expressed as

```
fact = snd o red[nat] Nat{c := nat * int;
                          $Zero := (Zero,1),
                          $Succ := \(m,n) (Succ m, ntoi(Succ m) * n)}
```

To define a general primitive recursion scheme for natural numbers, declare a combinator, *Pr*, by

```
type a;
Pr(g,h) = snd o red[nat] Nat{c := nat * a;
                             $Zero := (Zero,g),
                             $Succ := <Succ o fst, h>}
```

in which the angle brackets designate a functional pair, $\langle f, g\rangle x = (fx,\ gx)$. This defines a family of *nat* algebras, with structure functions $Pr(g,h) : nat \rightarrow a$, for each pair $(g : a,\ h : nat{*}a \rightarrow a)$. In terms of this scheme, the factorial function is defined by

```
fact = Pr(1,\(m,n) ntoi(Succ m) * n)
```

where the type variable a has been instantiated to *int*.

2.4.1 Generalized primitive recursion

The primitive recursive control scheme can be generalized to algebras of other varieties. There is no special combinator in ADL for primitive recursion, but a primitive recursion combinator can be composed for each variety. For example, we shall define a primitive recursion for *List*-algebras,

$$\begin{array}{lcl} Pr\_list & : & list(a) \rightarrow b \\ Pr\_list & = & snd\,(red[list]\,List\{c := list(a) \times b;\ \ \$Nil := g,\ \ \$Cons := f\}) \end{array}$$

where $g : (list(a) \times b)$ and $f : (a \times (list(a) \times b)) \rightarrow (list(a) \times b)$.

Exercise 2.2 Splitting a list

Define $splitat : char \rightarrow list(char) \rightarrow list(char) \times list(char)$

The function *splitat* is specified as follows:

If the list xs contains an occurrence of the character, c, then *splitat* c xs yields the pair of the prefix and suffix of the first occurrence of c in xs. Otherwise, it yields the pair (xs, Nil).

Hint: Use primitive recursion for *list*.

2.5 Proof rules for algebras

Inference rules for the particular algebras introduced in the previous section are summarized below. The rule for the *Nat*-algebra is natural induction, as one would expect. For the *List*, *Tree* and *Bush* algebras, the rules are those of "structural induction" for the datatypes that correspond to the free algebras. Each rule can be calculated from the signature of the variety whose homomorphisms it describes. Notice that we do not have to treat induction as a special rule of the logic—the inductive proof rules account for the computational content of the algebra homomorphisms. This has been noted previously by Goguen [Gog80] and others.

In the following rules, τ designates a type expression. In the last rule, the notation $y \in ys$ is the assertion that y is an element of the list ys.

$$\frac{\begin{array}{ll} f : \tau & P(f) \\ g : \tau \rightarrow \tau & \forall x : \tau.\, P(x) \Rightarrow P(g\, x) \end{array}}{\forall n : \mathit{nat}.\ P(\mathit{red}[\mathit{nat}]\ \mathit{Nat}\{c := \tau;\ \$\mathit{Zero} := f,\ \$\mathit{Succ} := g\}\, n)}$$

$$\frac{\begin{array}{ll} f : \tau & P(f) \\ g : a \times \tau \rightarrow \tau & \forall x : a. \forall y : \tau.\, P(y) \Rightarrow P(g(x, y)) \end{array}}{\forall y' : \mathit{list}(a).\, P(\mathit{red}[\mathit{list}]\ \mathit{List}(a)\{c := \tau;\ \$\mathit{Nil} := f,\ \$\mathit{Cons} := g\}\, y')}$$

$$\frac{\begin{array}{ll} f : a \rightarrow \tau & P(f\, x) \\ g : a \times \tau \rightarrow \tau & \forall y : \tau. \forall z : \tau.\, P(y) \wedge P(z) \Rightarrow P(g(y, z)) \end{array}}{\forall y' : \mathit{tree}(a).\, P(\mathit{red}[\mathit{tree}]\ \mathit{Tree}(a)\{c := \tau;\ \$\mathit{Tip} := f,\ \$\mathit{Fork} := g\}\, y')}$$

$$\frac{\begin{array}{ll} f : a \rightarrow \tau & P(f\, x) \\ g : \mathit{list}(\tau) \rightarrow \tau & \forall ys : \mathit{list}(\tau).\, (\forall y \in ys.\, P(y)) \Rightarrow P(g\,(ys)) \end{array}}{\forall z : \mathit{bush}(a).\, P(\mathit{red}[\mathit{bush}]\ \mathit{Bush}(a)\{c := \tau;\ \$\mathit{Leaf} := f,\ \$\mathit{Branch} := g\}\, z)}$$

3 Mutually recursive signatures

It is often convenient to define signatures by mutual recursion. For example, signatures that correspond to the syntactic phyla of a context-free grammar may be mutually recursive. Each signature is comprised of operators that correspond to the productions for a single phylum. Mutually recursive signatures can be described in terms of multi-sorted algebras.

ADL supports the definiton of a finite set of mutually recursive signatures called a *family.* A family declaration contains a list of sorts with their arities, followed by a sequence of signatures named by these sorts.

Example 3.1 The following two-sorted signature corresponds to an abstract syntax of arithmetic terms, stratified by additive and multiplicative operators:

```
signature TermGram(a){type c, d;
                   term/c = {$Add of c*c,
                                $Neg of c,
                                $Prim of d}
                   factor/d = {$Mpy of d*d,
                                  $Subterm of c,
                                  $Ident of a}}
```

The operators *$Prim* and *$Subterm* coerce a value from a representation in one carrier to a representation in the other.

□

The reduction combinator accommodates multi-sorted signatures by using the family of sort names as an index set. It takes a sort and a sort-indexed family of algebras as its arguments, and yields a structure function for the specified sort in this particular algebra. The sort given as the first argument of the combinator determines the typing of the combinator expression.

$$
\begin{array}{rcl}
\mathit{red}[\mathit{term}]\ \mathit{TermGram}(a)\{c := t_1,\ d := t_2;\ \mathit{term}\{\ldots\}\ \mathit{factor}\{\ldots\}\} & : & \mathit{term}(a) \rightarrow t_1 \\
\mathit{red}[\mathit{factor}]\ \mathit{TermGram}(a)\{c := t_1,\ d := t_2;\ \mathit{term}\{\ldots\}\ \mathit{factor}\{\ldots\}\} & : & \mathit{factor}(a) \rightarrow t_2
\end{array}
$$

Example 3.2 To define an arithmetic calculator, where the type *a* is *string*, apply *red*[*term*] to the following *TermGram*-algebra:

```
TermGram(int){c := int; d := int; term{$Add := (+), $Neg := ~, $Prim := id},
              factor{$Mpy := (*), $Subterm := id, $Ident := string_to_int}}
```

□

Exercise 3.1 Printing the image of a term

Define a print function to print values of type *term*(*int*) as strings of **ascii** characters, with infix operator symbols for *add* and *mpy*, using an implicit operator precedence and with the minimum number of parentheses necessary to avoid incorrect associations. Assume that *add* and *mpy* are associative operators.

3.1 Proof rules for multi-sorted algebras

Proof rules for a multi-sorted variety are derived from its signature in a manner analogous to that for a single-sorted variety. There is a distinct "result" predicate symbol for each sort in the family. The predicate indexed by a particular sort must hold for all expressions typed in the carrier of that sort. The hypotheses of a proof rule are implications that assure that the application of each operator satisfies the result predicate associated with its sort. A rule has a consequent for each sort in the family.

Example 3.3 Proof rule for *TermGram*

For the signature given in Example 3.1 it is straightforward to derive a proof rule with two consequents. Let the symbols P and Q designate the result predicates for sorts *term* and *factor*, respectively. Let τ_1 and τ_2 denote types. The rule is:

$$\begin{array}{ll}
p : \tau_1 \times \tau_1 \rightarrow \tau_1 & \forall x_1, x_2 : \tau_1.\, (P(x_1) \land P(x_2)) \Rightarrow P(p\,(x_1, x_2)) \\
n : \tau_1 \rightarrow \tau_1 & \forall x : \tau_1.\, P(x) \Rightarrow P(n\,x) \\
q : \tau_2 \rightarrow \tau_1 & \forall y : \tau_2.\, Q(y) \Rightarrow P(q\,y) \\
m : \tau_2 \times \tau_2 \rightarrow \tau_2 & \forall y_1, y_2 : \tau_2.\, (Q(y_1) \land Q(y_2)) \Rightarrow Q(m\,(y_1, y_2)) \\
k : \tau_1 \rightarrow \tau_2 & \forall x : \tau_1.\, P(x) \Rightarrow Q(k\,x) \\
j : a \rightarrow \tau_2 & \forall u : a.\, Q(j\,u) \\
\hline
\multicolumn{2}{l}{\forall z : \mathit{term}(a).\, P(\mathit{red}[\mathit{term}]\ \mathit{TermGram}(a)\{c := \tau_1,\ d := \tau_2;\ \mathit{term}\{\$\mathit{Add} := p,\ \$\mathit{Neg} := n,\ \$\mathit{Prim} := q\}} \\
\multicolumn{2}{r}{\mathit{factor}\{\$\mathit{Mpy} := m,\ \$\mathit{Subterm} := k,\ \$\mathit{Ident} := j\}\}\ z)} \\
\multicolumn{2}{l}{\forall z : \mathit{factor}(a).\, Q(\mathit{red}[\mathit{factor}]\ \mathit{TermGram}(a)\{c := \tau_1,\ d := \tau_2;\ \mathit{term}\{\$\mathit{Add} := p,\ \$\mathit{Neg} := n,\ \$\mathit{Prim} := q\}} \\
\multicolumn{2}{r}{\mathit{factor}\{\$\mathit{Mpy} := m,\ \$\mathit{Subterm} := k,\ \$\mathit{Ident} := j\}\}\ z)}
\end{array}$$

Although this rule may appear formidable in terms of the sheer volume of symbols, its structure is absolutely straightforward and unsurprising. It expresses the structural induction entailed by the signature *TermGram*.

4 Morphisms of non-initial structure algebras

Recall the diagram in terms of which a T-algebra morphism is defined:

$$\begin{tikzcd}
t(a) \arrow[r, "map\_t\,f"] \arrow[d, "h"'] & t(b) \arrow[d, "k"] \\
a \arrow[r, "f"] & b
\end{tikzcd}$$

The initial algebra homomorphisms illustrated in the diagram are h, k, $map\_t\,f$ and the NW-to-SE diagonal arrow. The homomorphism condition of the diagonal is the commutation condition of the diagram,

$$f \circ h \quad = \quad k \circ map\_t\,f \tag{1}$$

Each homomorphism can be expressed in terms of the combinator *red* and the appropriate T-algebra. However, f is also a T-algebra morphism, and under certain conditions, it may also be expressed in terms of a combinator.

Suppose there were a function $p : a \rightarrow t(a)$ such that $p \circ h = id_{t(a)}$. This function need not be a right inverse for h on a, but it is a right inverse when restricted to a domain that is the h-image of $t(a)$,

$$h \circ p = id_{a \downarrow h}$$

where the notation $a \downarrow h$ means the set a restricted to the codomain of h. Post-composing both sides of (1) with p gives an equation satisfied by f when its domain is restricted to the h-image of $t(a)$:

$$f = k \circ map\_t\,f \circ p \tag{2}$$

This equation suggests a way to realize f as the composition of a homomorphism with a function that analyzes the domain $a \downarrow h$.

Let $E\$t(a)$ designate a type isomomorphic to $t(a)$. Typically, it will be a disjoint union of alternatives including a, $t(a)$, the "unit" type, $\mathbf{1}$, and products of these. It represents an explicit one-level unfolding of the structure of terms of type $t(a)$. Then a function $p' : a \rightarrow E\$t(a)$ may be isomorphic to a left inverse for h as described in the preceding paragraph. With this nomenclature, an isomorphic relative of equation (2) given above can be summarized in the diagram below, which reveals the structure more clearly:

$$\begin{tikzcd}
E\$t(a) \arrow[r, "map\_E\$t\, f"] & E\$t(b) \arrow[d, "k"] \\
a \arrow[u, "p'"] \arrow[r, "f"'] & b
\end{tikzcd}$$

Following the suggestion outlined above, ADL introduces a combinator with which to construct morphisms whose domains are T-algebras that are not initial. We call this combinator *hom* . It takes three parameters;

1. the sort of the structure function that is to be mapped,

2. the structure algebra in the codomain of the morphism and

3. a partition relation that is the "inverse" structure function of its domain algebra.

The partition relation is typically expressed as a conditional or a *case* expression that tests a value of type a to reveal the structure of the algebra. The codomain of the partition relation is $E\$t(a)$, which is a disjoint union of the domain types of the set of operators of the signature T.

Thus we write $\mathrm{hom}[t]\,T\{b;\, k\}\,p$, where $k : t(b) \rightarrow b$ and $p : a \rightarrow t(a)$. Here is an example that illustrates the construction of a T-algebra morphism using *hom* .

Example 4.1 : Calculate the largest power of 2 that factors a given positive integer.

Consider the *Nat*-algebra defined by:

$$Nat\{c := int;\ \$Zero := m,\ \$Succ := \lambda n\, 2 \times n\}$$

in which the free variable m represents an odd, positive integer. The carrier of this algebra is the set consisting of $\{m, 2m, 4m, 8m, \ldots\}$. To invert the structure function, construct a function that recovers the natural number giving the power of two that multiplies m in forming any element of the carrier. That is, let

```
p : int → E$nat(int)
p =def \n  if n mod 2 <> 0 then $Zero
           else $Succ(n div 2)
```

where *E$nat* is a derived, unsaturated sort. This sort belongs to no declared variety, thus has no signature and cannot form the type of the domain or codomain of other, explicitly defined functions.

Notice that in the above definition, the operators of the *Nat*-algebra, *$Zero* and *$Succ*, assume specific types by binding the carrier as *int*. These occurrences of *$Zero* and *$Succ* represent the operators of the particular *Nat*-algebra that structures the *int*-typed domain of the *Nat*-algebra morphism being defined.

To complete the solution of the problem, we need to specify a *Nat*-algebra that yields an integer representation of a power of 2. To give an exponent of two, we can use the algebra in which a natural number is represented as a positive integer. This algebra was used to specify the function ntoi in an earlier example. (Notice that the bindings given to the operator symbols *$Zero* and *$Succ* in this algebra are not the same as the bindings presumed in in the definition of p above. In general, they need not even have the same typings.) Thus, we get an algorithm expressed in ADL as:

```
pwr_2 = hom[nat] Nat{c := int; $Zero := 0, $Succ := \n 1+n} p
```

The equation satisfied by *pwr_2* is:

$$\begin{aligned} pwr\_2\, n = &\ \textbf{if}\ n \bmod 2 \neq 0\ \textbf{then}\ 0 \\ &\ \textbf{else}\ 1 + pwr\_2\,(n\ \mathrm{div}\ 2) \end{aligned}$$

To obtain an explicit representation of the factor that is a power of 2, the *Nat*-algebra can be modified to calculate that factor. This solution is

```
pwr_2' = hom[nat] Nat{c := int; $Zero := 1, $Succ := \n 2*n} p
```

Example 4.2 : $\log_2$ of a positive integer.

By modifying the algebra in the domain of the partition relation in Example 4.1 we can obtain an algorithm for the base 2 logarithm of a positive integer. Let

$$p' : \mathit{int} \rightarrow \mathit{E\$nat(int)}$$

```
p' =def \n  if n div 2 = 0 then $Zero
            else $Succ(n div 2)
```

```
log_2 = hom[nat] Nat{c := int; $Zero := 0, $Succ := \n 1+n} p'
```

Example 4.3 : Filtering a list

The function *filter p* : *list*(a) → *list*(a) reconstructs from a list given as its argument, a list of the subsequence of its elements that satisfy the predicate function p : a → *bool.* This function can be directly constructed as an instance of *red* for a suitable list algebra. However, we propose an algebraic variety to represent the two cases that occur in filtering—an element of the list is either to be included or omitted.

```
signature Slist(a){type c; slist/c = {$Nomore, $include of a*c, $omit of c}}
```

A definition of *filter p* can be given as a morphism of *Slist*-algebras:

```
filter p = hom[slist] Slist(a){c := list(a);
                                $Nomore = Nil,
                                $Include = Cons,
                                $Omit = id}
                    (\xs case xs of
                          Nil => $Nomore
                        | Cons(x,xs') => if p x then $Include(x,xs')
                                         else $Omit xs'
                        end)
```

□

Example 4.4 : Quicksort

A quicksort of a list of integers requires two functions, one that partitions a list,

$$part\ :\ int \longrightarrow list(int) \longrightarrow list(int) \times list(int)$$

and another that sorts a list, $sort\ :\ list(int) \longrightarrow list(int)$. The function *part* can be defined as a reduce:

```
part a = red[list] List(int){c := list(int)*list(int);
                             $Nil := (Nil,Nil),
                             $Cons := \(b,(xs,ys)) if b<a then (Cons(b,xs),ys)
                                                   else (xs,Cons(b,ys))}
```

The function *sort*, however, is a divide-and-conquer algorithm with the structure of a binary tree. It can be expressed as a hom of the following algebraic variety:

```
signature Btree(a){type c; btree/c = {$Emptytree, $Node of c*a*c}};
sort = hom[btree] Btree(int){c := list(int);
                             $Emptytree := Nil,
                             $Node := \(xs,x,ys) append xs (Cons(x,ys))}
                  (\xs case xs of
                         Nil => $Emptytree
                       | Cons(x,xs') =>
                                let (ys,ys') = part x xs'
                                in $Node(ys,x,ys')
                       end)
```

Notice that although the control is a tree traversal, the sort function has type $list(int) \rightarrow list(int)$. There is no data structure corresponding to the datatype $btree(list(int))$. This is a "treeless" tree traversal.

Exercise 4.1 Another form of bush

Given **signature** $Bush'(a)\{\text{type } c;\ bush'/c = \{\$Leaf' \text{ of } a,\ \$Branch' \text{ of } nat * (nat \to c)\}\}$ construct a morphism of type $bush(a) \to bush'(a)$ that is invertible. (Construct its inverse, too.)

Exercise 4.2 Splitting a list more efficiently

The function *splitat* defined by primitive recursion does more computation than is necessary. It recursively evaluates the function on the tail of a list that has already been successfully split. Reformulate the function as a *hom*[*list*].

Exercise 4.3 Factors of a positive integer

Give a function, *factors*, that takes a positive integer N and a list of positive integers M to a list of the factors of N by M, and which satisfies the following equations:

$$\begin{array}{ll} factors\ N\ \ Nil = Cons(N, Nil) & \\ factors\ N\ \ Cons(m, M') = Cons(m, factors\ (N/m)\ \ Cons(m, M')) & \text{if } m \text{ divides } N \\ factors\ N\ \ Cons(m, M') = factors\ N\ \ M' & \text{otherwise} \end{array}$$

Prove that your solution satisfies the equations.

4.1 Proof rules for morphisms of non-initial algebras

Properties of functions constructed with *hom* can be verified by applying the proof rules of the T-algebra, as described earlier, provided that the construction is actually a T-algebra morphism. Recall that for a construction $hom[\text{t}]\ T(a)\{b;\ k\}\ p$ to be a T-algebra morphism, the partition relation p must be a left inverse of the structure function of a T-algebra, (a, h). Since we do not know h in general, we require a condition that can be applied directly to p itself. Note that if p is a left inverse, it is also a right inverse to h on some subset of the elements of type a. Thus p is necessarily *formally* correct in the sense that it constructs results by well-typed application of operators of the signature T. However, its application to an arbitrary element $x : a$ might fail to be defined; x may not be in the codomain of h. The additional requirement can be stated in terms of a total ordering on a that must be provided to discharge the proof obligation.

Definition 4.1 Let T be a signature declared by

$$\textbf{signature } T(a)\{\textbf{type } c;\ s/c = \{\ldots\ \kappa_i \textbf{ of } t_{i1} \times \cdots \times t_{im_i}\ \ldots\}\}$$

Let P be a predicate over a. Suppose that $(\prec) \subseteq a \times a$ is a well-founded ordering on the set $\{x : a \mid P(x)\}$. We say that a function $p : a \rightarrow s(a)$ calculates a *T-inductive partition* of the set $\{x : a \mid P(x)\}$ if

$$\begin{array}{l} \forall x : a.\ P(x) \Rightarrow \\ \qquad \forall \kappa_i \in T.\, p\, x = \kappa_i(y_1, \ldots, y_{m_i}) \Rightarrow \\ \qquad\qquad \forall j \in 1..m_i \left\{ \begin{array}{ll} y_j \prec x & \text{if } t_{ij} = c \\ \forall z.\ z\ \mathit{elt\_s'}\ y_j \Rightarrow z \prec x & \text{if } t_{ij} = s'(c) \end{array} \right. \end{array}$$

where s' is an unsaturated sort ($s' \neq s$) and *elt_s'* is an infix notation for the two-place predicate defined by:

$$\begin{array}{rcl} z = x & \Rightarrow & z\ \mathit{elt\_s'}\ \kappa'_i(y_1, \ldots, x, \ldots, y_{m_i}) \\ z\ \mathit{elt\_s'}\ y & \Rightarrow & z\ \mathit{elt\_s'}\ \kappa'_i(y_1, \ldots, y, \ldots, y_{m_i}) \end{array}$$

for all operators κ'_i in the signature of sort s'.

□

In the definition above, the predicate P characterizes a subset of type a elements on which the morphism is well-defined. Any properties of the morphism deduced with the proof rules of the T-algebra will be valid only for points of the domain that satisfy P. In Example 4.1, a suitable subset and its well-ordering is the positive integers with the natural order, $(<)$. The partition relation p induces a *Nat*-inductive partition on this subset. In Example 4.2, the same ordering is used but the set is the non-negative integers. In Examples 4.3 and 4.4, a suitable ordering on *list*(*int*) is $xs \prec ys$ iff $\mathit{length}\ xs < \mathit{length}\ ys$. The verification condition for the function *part* of the *Quicksort* example becomes

$$xs = \mathit{Cons}\,(x, xs') \wedge \mathit{part}\, x\ xs' = (ys, ys') \Rightarrow ys \prec xs \wedge ys' \prec xs$$

Definition 4.1 of T-inductive partition of a set extends without complication to algebras of a multi-sorted signature. What becomes more complicated in such a case is the well-founded order, which may relate terms of different sorts.

5 The ADL type system

Logical properties of structure-algebra morphisms can be derived by inductive proof rules. Each such property can be formalized as a predicate over a set. ADL types can be interpreted as sets, although as we shall see, when the *hom* combinator is introduced, proof obligations arise in verifying that a syntactically legal term is semantically valid with respect to the ADL type system.

Since types are sets, the restriction of a type by a predicate defines a set that may be considered to be a subtype of a structurally defined type. We call such subtypes *domain types*. An ADL domain type is expressed with set comprehension notation, as for instance, $\{x : \tau \mid P(x)\}$, where τ is a structural type expression and P stands for a predicate. In the type system of ADL, domain types occur only on the left of the arrow type constructor. Domain types express restrictions in the types of functions.

Syntax of type expressions

```
typ      ::=  identifier                             primitive types
           |  typ * typ                              products
           |  domtyp → typ                           function types
           |  identifier(typ [, typ])                datatypes

domtyp   ::=  typ
           |  {identifier : typ | Identifier(expr)}  restricted domain types
```

The Hindley-Milner type system is based upon a structural notion of type and is not expressive enough to distinguish among domain types of ADL. Thus, its type-checking algorithm is not powerful enough to ensure that a syntactically well-formed ADL expression is meaningful, but requires additional evidence as proof. Nevertheless, we find it useful to employ the Hindley-Milner type system as an approximation to ADL's type system. The Hindley-Milner type-inference algorithm is an abstract interpretation of ADL that approximates its type assignments. Whenever Hindley-Milner type checking asserts that an expression is badly typed, it cannot be well-typed in the ADL type system. When Hindley-Milner type inference assigns a type to an expression, that typing will be structurally compatible with any ADL typing of the expression.

For example, given a pair of ADL functions with typings $f : \{x : \tau_1 \mid P(x)\} \rightarrow \tau_2$ and $g : \{x : \tau_2 \mid Q(x)\} \rightarrow \tau_3$, a structural (Hindley-Milner) typing approximates the ADL typings as $f : \tau_1 \rightarrow \tau_2$ and $g : \tau_2 \rightarrow \tau_3$. It will judge their composition to be well-typed, with typing $g \circ f : \tau_1 \rightarrow \tau_3$. An ADL typing of the composition has the form $g \circ f : \{x : \tau_1 \mid R(x)\} \rightarrow \tau_3$, and it carries a proof obligation to show that $R(x) \Rightarrow P(x) \wedge Q(f\,x)$. To discharge the proof obligation requires a logical deduction based upon algebric properties of the function f.

To determine whether a function application is well-typed is too complex for Hindley-Milner typing alone. To know that $f\,a$ is well-typed, one must furnish evidence that $P(a)$ holds. Function types in ADL may involve restrictions expressed in domain types, and these restrictions might include arbitrary arithmetic formulas. For this reason, ADL does not have principal types, nor unicity of types. Domain restrictions are needed to express the termination conditions for combinators that express morphisms of non-initial structure algebras.

Domain restrictions must be expressible with first-order predicates. As a practical consequence, this implies that a domain restriction cannot assert a property of the result of applying a function-typed variable. For example, given a function $f : \{x : \tau_1 \mid P\,x\} \rightarrow \tau_2$, we can express the typing of a function that composes its argument on the left of f as

$$\lambda g.\, g \circ f : (\tau_2 \rightarrow \tau_3) \rightarrow \{x : \tau_1 \mid P\,x\} \rightarrow \tau_3$$

The type of the formal parameter, g, is only structural; it requires no domain predicate to be imposed.

If, however, we attempt to type the function $\lambda h.\, f \circ h$ that composes its argument on the right of f, we find that it is impossible to do so with only a first-order domain predicate. The predicate must express that every point in the codomain of h satisfies the domain predicate P, and to express this restriction requires quantification over all points in the domain of h. The only kind of typing restriction that can be expressed of a function-typed variable is a domain restriction. Nevertheless, this can be quite powerful.

Given a proof that a function-typed variable satisfies a domain restriction at every occurrence in an expression, the variable may be abstracted from the expression and given a domain-restricted function type. For instance, suppose that in an expression $\lambda x.\, e : \tau_1 \rightarrow \tau_3$, the free variable f occurs in an applicative position and satisfies a structural typing $f : \tau_1 \rightarrow \tau_2$. If in addition, at every occurrence of f in e (each of the form $f\, e'$) one can show that $P\, x \Rightarrow R\, e'$, then the abstraction can be given a typing $\lambda f.\, \lambda x.\, e \ :\ (\{y : \tau_1 \mid R\, y\} \rightarrow \tau_2) \rightarrow \{x : \tau_1 \mid P\, x\} \rightarrow \tau_3$.

An application of a function $h \ :\ (\{y : \tau_1 \mid R\, y\} \rightarrow \tau_2) \rightarrow \tau_3$ to an argument $e' \ :\ \{y : \tau_1 \mid Q\, y\} \rightarrow \tau_2$ is judged to be well-typed if there is a proof that $\forall y : \tau_1.\, R\, y \Rightarrow Q\, y$. This condition ensures that any expression to which h's formal parameter might be applied will satisfy the restriction imposed by the domain predicate in the typing of e'.

5.1 Typing combinator expressions

The function composition operator is one instance of an ADL combinator whose arguments can have domain-restricted function types. The ADL combinators *red* and *hom* are further instances, and they require special typing rules. These combinators are applied to algebra specifications, so it is necessary to specify what constitutes a well-typed algebra specification. For simplicity, we illustrate the formal rules for a single-sorted algebra A, with sort symbol s and carrier (type) symbol c. Let $Index(\Sigma_s)$ designate the set of indices that enumerate the signature of sort s. Let $\tau, \tau_1, \tau_2, \cdots$ range over types and $f_1, f_2, \cdots$ range over expressions. Let ρ range over typing environments. (A typing environment is a finite mapping of type variables to types.) The judgement form $\rho \vdash e : \tau$ is read as "expression e has type τ in the typing environment ρ." The rule for well-typing of an algebra specification is:

$$\frac{\begin{array}{l}\forall i \in Index(\Sigma_s).\ [a : \text{type}], \rho \vdash f_i \ :\ \tau_i \rightarrow \tau \\ \qquad\qquad\quad \wedge\ \tau_i = \rho_c[\tau/c](\sigma_i) \quad \text{where } (\kappa_i, \sigma_i) \in \Sigma_s\end{array}}{[a : \text{type}], \rho \vdash_{Alg} A(a)\{c := \tau,\ s\{\ldots \kappa_i := f_i, \ldots\}\}}$$

in which ρ_c is the type environment that agrees with ρ on all type variables except c, which is not in its domain.

Well-typing of an algebra specification is a hypothesis for the typing of a reduce combinator. The rule is:

$$\frac{[a : \text{type}], \rho \vdash_{Alg} A(a)\{c := \tau,\ s\{\ldots \kappa_i := f_i, \ldots\}\}}{[a : \text{type}], \rho \vdash \mathit{red}[s]\, A(a)\{c := \tau,\ s\{\ldots \kappa_i := f_i, \ldots\}\} \ :\ s(a) \rightarrow \tau}$$

To type instances of non-initial algebra morphisms, we require a typing for partition relations. The codomain of a partition relation does not have a unique structural type, for it is only specified up to a variety. To express this, ADL provides a unique type constructor, $E\$s(a)$, to correspond to each (unsaturated) sort, s. The type of a partition relation for this sort will be of the form $\tau' \rightarrow E\$s(\tau')$, where τ' is a type bound to the carrier of the domain algebra for an instance of hom[s]. The well-typing of a partition relation furnishes an additional hypothesis of the typing rule for hom.

$$\frac{\begin{array}{l} p \,:\, \tau' \rightarrow E\$s(\tau') \\ [a : \text{type}],\, \rho \vdash_{Alg} A(a)\{c := \tau,\ s\{\ldots \kappa_i := f_i, \ldots\}\} \end{array}}{\rho \vdash \text{hom}[s]\, A(a)\{c := \tau,\ s\{\ldots \kappa_i := f_i, \ldots\}\}\, p \,:\, \tau' \rightarrow \tau}$$

For an example, consider typing the definition of *pwr_2* in Example 4.1. First, we check the well-typing of the *nat* algebra. For the carrier binding $c := int$, the operator typings will be $\$Zero : int$ and $\$Succ : int \rightarrow int$. These are satisfied by the bindings $\$Zero := m$ and $\$Succ := \lambda n\, n + 1$, where $m : int$. Thus the algebra specification $Nat\{c := int;\ nat\{\$Zero := m,\ \$Succ := \lambda n\, n + 1\}\}$ is well-typed.

Next we type the partition relation, p. In this relation, the operators *$Zero* and *$Succ* are considered to be unbound, and so their typings are expressed with the codomain type represented by $E\$nat(a)$, Choosing $a = int$ we get the specific typings $\$Zero : E\$nat(int)$ and $\$Succ : int \rightarrow E\$nat(int)$, which gives p the typing $p : int \rightarrow E\$nat(int)$. Applying the rule for structural typing of hom gives

$$pwr\_2 = \text{hom}[nat]\ Nat\{c := int;\ nat\{\$Zero := m,\ \$Succ := \lambda n\, n + 1\}\}\ p \ :\ int \rightarrow int$$

However, to get the proper ADL typing, we must provide a domain predicate under which the algorithm can be proved to terminate. A termination condition is that the operation $\lambda n\, n$ **div** 2 must be compatible with a well-ordering relation over the predicated domain. A suitable domain restriction is $\forall n : int.\, n \neq 0$. Thus a proper ADL typing is

$$pwr\_2 \ :\ \{n : int \mid n \neq 0\} \rightarrow int$$

This typing is not unique, however. Another proper typing is

$$pwr\_2 \ :\ \{n : int \mid n > 0\} \rightarrow int$$

6 Data and codata

So far, we have only considered signatures of algebraic varieties, in which each operator has a typing of the form $op_i : \tau_{i,1} * \ldots * \tau_{i,m_i} \rightarrow c$, where c designates the type of the carrier. There is a dual to this construction.

Suppose a signature (of a single sort) were to consist of typings of the form $op_i : c \rightarrow \tau_{i,1} * \ldots * \tau_{i,m_i}$, where c might occur in the type expressions $\tau_{i,j}$. Such a collection of operators is analogous to the projection functions of a *record* template. A signature of this form specifies a covariety of coalgebras. The definition of a coalgebra is dual to that of a structure algebra.

Definition 6.1 Let t denote an unsaturated sort of a (single-sorted) coalgebra signature, T. A *T-coalgebra* consists of a pair (c, k) where c is a set, the carrier of the coalgebra, and $k : c \rightarrow t(c)$ is called its co-structure function.

□

This very general definition reveals the essential information that a coalgebra is:

- a structure that is defineable over sets,
- composed of a carrier set and a structure function that is total.

This definition can be generalized to account for multi-sorted signatures by indexing the algebra components with sorts, i.e. $\{(c_t, k_t)\}$, where t ranges over the finite set of sorts defined in the signature. The use of a saturated sort expression, $t(a)$, to represent a type (set) in the definition tells nothing about the permissible structure of that set. We shall be interested in signatures for which the interpretation of $t(a)$ is a cartesian product based upon the types a and $c_{\tau_1}, \ldots, c_{\tau_n}$, the carriers of the n sorts of T (and on combinations of these types called *tensors*, which is a topic that is beyond the scope of this document).

Among the coalgebras that share a common signature, there is one for which the projectors are free, unconstrained by any theory. For the free coalgebra, we may think of the the carrier as the set of (infinite) records from which independent values are projected by repeated applications of the projectors. In general, the projectors of a coalgebra should be thought of as witnesses of the carrier. Coalgebras play as significant a role in ADL as do algebras. They define iterative control structures.

Elements of the carrier of a coalgebra are in general, infinitary objects and thus, are not to be confused with ordinary data. The projectors of a coalgebra afford the only means to access these objects to produce data. We refer to the elements of a coalgebra's carrier as *codata* and to the type of a carrier as a *cotype.* Thus, since both algebras and coalgebras are present in ADL, its type system is bifurcated into two sorts: data and codata types.

The quintessential covariety of coalgebras has the following signature:

$$\textbf{cosignature}\ Stream(a)\{\textbf{type}\ c;\ str/c = \{\$Shd\ :\ a,\ \$Stl\ :\ c\}\}$$

The carrier of a free *Stream* coalgebra is a codata type $str(a)$, whose elements are infinite sequences, or streams, of homogeneously typed elements. The free projectors $Shd : str(a) \rightarrow a$ and $Stl : str(a) \rightarrow str(a)$ provide the sole means by which to witness a stream. Every stream is infinite; that is, it is always meaningful to apply the projectors to a stream, even though there is no way to witness the entire stream at once.

A stream provides a good model for an incrementally readable input file. The projection *Shd* yields the value of the first element of a stream, just as a *get* operation on an open file produces a value from the file buffer. However, the file primitive, *get*, advances a buffer pointer as a side effect, so that the next invocation of *get* on the same file will yield the next element. Taking a *Shd* projection implies no side effect. A separate projection, *Stl*, yields the tail of a stream but it is not manifested until projections of it are taken. The situation is familiar in languages with non-strict operators.

There are both finite and infinite access paths to elements of a stream. A path is expressed by a well-typed composition of the projectors *Shd* and *Stl.* Every finite path ends in *Shd.* To be able to generate every path in a stream, a control structure must support repeated applications of *Stl* until there is a final application of *Shd*, which terminates the path.

6.1 Control structures induced by coalgebras

Useful control structures that can be derived from coalgebras. The mathematical notion underlying these control structures is that of a morphism of coalgebras, of a given covariety.

Definition 6.2 Let T be a single-sorted covariety, with an unsaturated sort symbol t. A function $g : a \rightarrow b$ is a *T-coalgebra morphism* if there are T-coalgebras (a, h) and (b, k) such that the following square commutes:

$$\begin{tikzcd}
t(a) \arrow[r, "map\_t\ g"] & t(b) \\
a \arrow[u, "h"] \arrow[r, "g"'] & b \arrow[u, "k"]
\end{tikzcd}$$

□

Like Definition 6.1, the definition of coalgebra morphism can also be extended to multi-sorted coalgebras.

A T-coalgebra *generator* is a function of a type $a \rightarrow t(b)$ that can be defined as the composition of a T-coalgebra structure function with a T-coalgebra homomorphism. That is, it corresponds to a diagonal arrow in the diagram above. A generator is characterized by a coalgebra specification in ADL. A coalgebra specification is an instance of a coalgebra signature, and provides a type for the carrier and specific functions for the projectors of the signature. However, the type parameter of an unsaturated sort, t, is not restricted to be the same as the carrier, as is the case in Definitions 6.1 and 6.2.

The construction of a generalized co-structure function can be understood in terms of the diagram below, which represents one level of "unfolding" of the definition of a free coalgebra. To express the unfolding, we require some notation for the general case of the coalgebraic structure expressed in a signature. For simplicity, we consider a single-sorted signature whose sort is unsaturated, i.e. $t : * \rightarrow *$. Suppose the signature consists of n projectors. The i^{th} projector has a typing $c \rightarrow \tau_{i,1} * \ldots * \tau_{i,m_i}$ where c is the type variable representing the carrier,

a is the type variable representing the type parameter of the sort, and each of the $\tau_{i,j}$ is either c or a or a type formed of these. We can represent the type of the i^{th} projector by $c \rightarrow F_i(a,c)$, capturing with the symbol F_i the structure of the i^{th} codomain type. We shall also use the symbol F_i to designate a combinator that constructs a function $F_i(u,v) : F_i(a,c) \rightarrow F_i(a,t(a))$ by the component-wise application of $u : a \rightarrow a$ to each component of type a, and $v : c \rightarrow t(a)$ to each component of type c. We designate the component-wise application by $F_i(u,v)$, for each $i \in 1..n$. This notational convention is used in the following diagram, in which $\times^n$ means the n-fold product of the indexed family of components.

$$\begin{tikzcd}
a \arrow[r, "g"] \arrow[d, dotted, "k"'] & \times^n F_i(b,a) \arrow[d, "\times^n F_i(id_b,k)"] \\
t(b) \arrow[r, "\mathbf{out}"'] & \times^n F_i(b,t(b))
\end{tikzcd}$$

In the diagram, **out** designates a natural (i.e. polymorphic) isomorphism. The dotted arrow is uniquely determined by the other data in the diagram, which is to say that the diagram constitutes a definition for the function indicated by the dotted arrow. Since **out** is an isomorphism, it has an inverse, designated by $\mathbf{out}^{-1}$. The generalized co-structure function, k, satisfies the equation

$$k = \mathbf{out}^{-1} \circ \times^n F_i(id, k) \circ g$$

The data on which k depends consists of the sort, t, and the coalgebra specified by g. In ADL, a generalized co-structure function is defined in terms of a combinator *gen* applied to these data.

Example 6.1 For example, the following expression generates a stream of ascending integers from an integer given as its argument:

$$\mathit{from} = \mathit{gen}[\mathit{str}]\, \mathit{Stream}(\mathit{int})\{c := \mathit{int};\ \$\mathit{Shd} := \mathit{id},\ \$\mathit{Stl} := \mathit{add}\,1\}$$

Thus *from* 0 generates the sequence of non-negative integers. We have the following equalities:

$$\mathit{Shd}(\mathit{from}\,0) \quad = \quad 0$$

$$\begin{aligned} Stl(from\,0) &= from\,1 \\ Shd(Stl(from\,0)) &= 1 \\ Stl(Stl(from\,0)) &= from\,2 \\ \cdots &= \cdots \end{aligned}$$

From these equalities we see that every finite path of the stream *from* 0 can be witnessed. Note that the witnesses gotten by applying *Stl* are typed as codata, with the consequence that these computations are suspended. Elements of a codata type are not proper values.

□

Example 6.2 A stream constructor function can be defined in terms of the stream generator combinator and the first and second projections of a cartesian product. Cartesian products exist in ADL because all functions are total. A polymorphic stream constructor $str\_cons : a \times str(a) \rightarrow str(a)$ is:

$$str\_cons = gen[str]\; Stream(a)\{c := a \times str(a);\; \$Shd := fst,\; \$Stl := snd\}$$

□

6.2 Witness paths

To compose a witness function for a coalgebra, we can formulate an inductive algebra to characterize a *path grammar* for the coalgebra. A path grammar generates all instances of finite paths, or witness functions, for the coalgebra. A path grammar is a meta-language concept, and is not formally expressible in ADL. Path grammars are useful for reasoning about coalgebras, however.

For *Stream*-coalgebras, all paths are linear, formed by iteration of the *Stl* projector. Thus a path grammar for *Stream* may be expressed as an instance of a *Nat* algebra whose carrier is a function from $str(\alpha)$ to the set of terms produced by witnessing a stream. We call this set of terms $L(str(\alpha))$. It corresponds to the union of the codomains of all the witness functions in the signature of the coalgebra. This union of codomains is not expressible as a type in ADL.

$$paths[str] = red[nat]\; Nat\{c := str(\alpha) \rightarrow L(str(\alpha));\; \$Zero := \$Shd,\; \$Succ := \lambda s.\, s \circ \$Stl\}$$

A natural number determines a path for $str(\alpha)$, and hence a witness function. For example,

$$Shd\,(paths[str]\;Succ(Succ(Zero))\;(from\;0)) = 2$$

For other coalgebras, the path grammars have more complex structure than *Nat* algebras.

Example 6.3 A coalgebraic variety with particularly simple structure is given by the signature

$$\textbf{cosignature}\; Iter\{\textbf{type}\; c;\;\; inf/c = \{\$Step\; :\; c\}\}$$

A generator for this coalgebra calculates a least fixpoint of the function bound to *\$Step*. Let $f\;:\;a \rightarrow a$. Then

$$fix\; f = gen[inf]\; Iter\{c := a;\;\; \$Step := f\}$$

This generated object is not only infinitary, it has no finite witness paths. However, this does not necessarily mean that it is devoid of computational meaning. If $a = b \rightarrow d$, then $fix\; f$ has an interpretation as the least fixpoint of f, in a domain of partial functions. However, $fix\; f$ is typed as *inf* by the structural typing rules of ADL, not as $b \rightarrow d$, hence no application of $fix\; f$ is well typed.

□

Example 6.4 Another interesting coalgebraic variety is expressed by the ADL declaration:

$$\textbf{cosignature}\; BinTree(a)\{\textbf{type}\; c;\;\; bintree/c = \{\$Val\; :\; a,\;\; \$Left, \$Right\; :\; c\}\}$$

A generator for *bintree*(*int*) is:

$$\begin{aligned} gen[bintree]\; BinTree(int)\{c &:= int; \\ \$Val &:= id, \\ \$Left &:= \lambda m\; 2m, \\ \$Right &:= \lambda m\; 2m + 1\} \end{aligned}$$

When this generator is applied to the integer value 1, it generates the infinite, binary tree whose integer labels enumerate the tree breadth-first.

A *bintree* generator incrementally generates streams of data witnessed via a sequence of binary decisions. The generator

$$\begin{aligned} \mathit{gen}[\mathit{bintree}]\ \mathit{BinTree}(\mathit{bool})\{c := \mathit{bool};\ & \\ & \$\mathit{Val} := \mathit{id}, \\ & \$\mathit{Left} := \lambda s\ \mathit{ff}, \\ & \$\mathit{Right} := \lambda s\ \mathit{tt}\} \end{aligned}$$

(where *tt* and *ff* are the identifiers of the boolean constants) generates a tree whose paths correspond to finite and infinite sequences of boolean-valued labels. Thus the set of paths contains the set of rational fractions in a binary representation. Its path grammar is specified with a *list*(*bool*) algebra.

$$\begin{aligned} \mathit{paths}[\mathit{bintree}] = \mathit{red}[\mathit{list}]\ \mathit{List}(\mathit{bool})\{c := \ & \mathit{bintree}(a) \rightarrow L(\mathit{bintree}(a)); \\ & \$\mathit{Nil} := \mathit{id}, \\ & \$\mathit{Cons} := \lambda(b, s).\ \textbf{if}\ b\ \textbf{then}\ s \circ \mathit{Right}\ \textbf{else}\ s \circ \mathit{Left}\} \end{aligned}$$

□

6.3 Proof rules for generators

The proof rules for generators assert properties that can be witnessed on all paths by which values of a coinductively generated object are accessed. These rules are somewhat weaker than the rules of coinduction, which allow the conclusion of assertions about codata [Pau93]. The proof rules given here are based upon induction over the set of witness paths.

For example, a proof rule for *Stream* generators is:

$$\frac{f : \tau \rightarrow \tau \qquad \forall x : \tau.\ P(x) \Rightarrow P(f\ x)}{\forall x : \tau.\ P(x) \Rightarrow \forall n : \mathit{nat}.\ \ P(\mathit{paths}[\mathit{str}]\ n\ (\mathit{gen}[\mathit{str}]\ \mathit{Stream}(t)\{c := \tau;\ \$\mathit{Shd} := \mathit{id},\ \$\mathit{Stl} := f\}\ x))}$$

For *BinTree* generators (Example 6.4), a proof rule is:

$$\frac{\begin{array}{ll} f : \tau \rightarrow \tau & \forall x : \tau.\ P(x) \Rightarrow P(f\ x) \\ g : \tau \rightarrow \tau & \forall x : \tau.\ P(x) \Rightarrow P(g\ x) \end{array}}{\begin{array}{l} \forall x : \tau.\ P(x) \Rightarrow \forall s : \mathit{list}(\mathit{bool}).\ \ P(\mathit{paths}[\mathit{bintree}]\ s \\ \qquad (\mathit{gen}[\mathit{bintree}]\ \mathit{BinTree}(t)\{c := \tau;\ \$\mathit{Val} := \mathit{id},\ \$\mathit{Left} := f,\ \$\mathit{Right} := g\}\ x)) \end{array}}$$

7 Constructing coalgebra morphisms

Generators have limited direct use as control paradigms for algorithms. Generators construct codata, which are the elements of so-called final data types [Hag87]. However, just as is the case for algebras, the morphisms of non-free coalgebras will yield many useful control schemes.

To calculate a coalgebra morphism it would suffice to have a left inverse to the arrow k on the right-hand side of the morphism diagram of Definition 6.2. Then an equation to be satisfied by a coalgebra morphism g can be read from the diagram below,

$$\begin{tikzcd}
\times^n F_i(a, t(a)) \arrow[r, "\times^n F_i(id_a,\, map\_t\, g)"] & \times^n F_i(a, t(b)) \arrow[d, "k^{-1} \circ \mathbf{out}^{-1}"] \\
a \arrow[u, "\mathbf{out} \circ h"] \arrow[r, dotted, "g"'] & b
\end{tikzcd}$$

The left inverse, $k^{-1} \circ \mathbf{out}^{-1}$, is called a *splitting function*, or splitter. It typically performs a conditional selection on some data projected from its argument. The equation

$$g = k^{-1} \circ \mathbf{out}^{-1} \circ \times^n F_i(id_a, map\_t\, g) \circ \mathbf{out} \circ h$$

is analogous to a recursive definition of g.

7.1 The combinator *cohom*

To realize morphisms of coalgebras that are not free, ADL provides a combinator, *cohom*, that takes as arguments a coalgebra and a splitting function. The splitting function specifies a path for witness of the coalgebra's carrier. The splitting function typically involves decisions conditional on witnessed values projected from the carrier, and thus, recursive elaboration of the path is implied. Just as was the case with morphisms of non-free algebras, there are proof obligations to show that the witness path elaborated in evaluating an instance of *cohom* is well-founded and hence, that the function is well-defined.

In implementing such a control structure, each application of an projector whose codomain includes the carrier, such as *$stl*, must be suspended or else the recursive elaboration of its repeated application would fail to terminate, affording no opportunity to witness finite paths.

To make effective the suspension of projections by projectors whose codomain type contains an instance of the carrier, any projection projector whose codomain includes the carrier is implicitly suspended in ADL. Suspension is not necessary for projections whose codomain type does not involve the carrier.

Example 7.1 For a familiar example of a *str*-algebra morphism, consider the construction from functions $p : c \to \mathit{bool}$ and $r : c \to c$,

$$\begin{array}{l} \mathit{while}_c\,(p,r) = \mathit{cohom}[\mathit{str}]\; \mathit{Stream}(a)\{c := a;\ \$\mathit{Shd} := \mathit{id}_a,\ \$\mathit{Stl} := r\} \\ \qquad\qquad\qquad\qquad (\lambda s.\ \textbf{let}\ x = \$\mathit{Shd}\,s\ \textbf{in} \\ \qquad\qquad\qquad\qquad\quad \textbf{if}\ p\,x\ \textbf{then}\ \$\mathit{Stl}\,s\ \textbf{else}\ x) \end{array}$$

This is the useful, unbounded iteration construct found in nearly all programming languages. It encompasses the paradigm of linear search. To be well-defined in ADL, a *while* iteration must be shown to be bounded. This question will be addressed when we consider proof rules and finiteness conditions for coalgebra morphisms.

Example 7.2 Another example of recursive stream generation is the following. Let $f : a \to a$. Define

$$\begin{array}{rcl} \mathit{rec}(f) & : & a \to \mathit{str}(a) \\ \mathit{rec}(f) & = & \mathit{cohom}[\mathit{str}]\; \mathit{Stream}(a)\{c := a;\ \$\mathit{Shd} := \mathit{id},\ \$\mathit{Stl} := \mathit{id}\} \\ & & \qquad\qquad (\lambda s\ \mathit{str\_cons}\,(\$\mathit{Shd}\,s,\ f\,(\$\mathit{Stl}\,s))) \end{array}$$

An equation that this coalgebra morphism satisfies is

$$\mathit{rec}(f) = \mathit{str\_cons} \circ \langle \mathit{id}_a,\ f \circ \mathit{rec}(f) \rangle$$

In this case we see that the codata generated is a stream of approximations to a limit of f. The limit itself will be the least fixed point of f.

□

Example 7.3 With a *bintree* morphism, we can specify binary search.

$$\begin{array}{l} \mathit{bsearch}\,(\mathit{key} : \mathit{int}) = \mathit{cohom}[\mathit{bintree}]\; \mathit{BinTree}(\mathit{int})\{c;\ \$\mathit{Val},\ \$\mathit{Left},\ \$\mathit{Right}\} \\ \qquad\qquad\qquad\qquad\qquad (\lambda s.\ \textbf{let}\ n = \$\mathit{val}\ s\ \textbf{in} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{if}\ n = \mathit{key}\ \textbf{then}\ n \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else if}\ n < \mathit{key}\ \textbf{then}\ \$\mathit{Left}\,s \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \$\mathit{Right}\ s) \end{array}$$

Here, the coalgebra specification is incomplete, as bindings for the carrier and the projectors have not been given. Binary search can be programmed for any *bintree* coalgebra whose witness function has *int* as its codomain.

Exercise 7.1 Consider an algebra of labeled, binary trees given by the following signature:

$$\begin{aligned}\textbf{signature } \mathit{Ltree}(a)\{c \text{ type};\\ \mathit{ltree}/c = \{\$\mathit{Empty},\\ \$\mathit{Node} \textbf{ of } c * a * c\}\}\end{aligned}$$

Give an algorithm in ADL to construct from an arbitrary instance of an *ltree* in the free term algebra, a new copy whose nodes are labeled by their enumeration in a breadth-first traversal. **Hint**: Assume that there is a stream of integers that are the generators for the labels to be used at each level in the tree. Use these to label the tree, then construct the stream with the paradigm of Example 7.2.

7.2 Typing coalgebra combinators

Like algebra specifications, coalgebra specifications also have simple structural typing rules. The rules presented in this report are restricted for simplicity to single-sorted coalgebras (with sort s and signature Σ_s). Typing rules for multi-sorted coalgbras are a straightforward extension to these rules.

The first rule is one for typing a coalgebra specification. The judgement form "$\rho \vdash_{\text{Coalg}} A\{\cdots\}$" can be read as "the coalgebra specification $A\{\cdots\}$ is well-typed relative to the environment ρ".

$$\frac{\begin{array}{l}\forall(\$\pi_i, \sigma_i) \in \Sigma_s.\ [a : \text{type}],\ \rho \vdash e_i : \tau \rightarrow \tau_i\\ \qquad\qquad \wedge\, \tau_i = \begin{cases}\rho[\tau/c](\sigma_i) & \text{if } c \text{ occurs in } \sigma_i\\ \rho(\sigma_i) & \text{otherwise}\end{cases}\end{array}}{[a : \text{type}],\ \rho \vdash_{\text{Coalg}} A(a)\{c := \tau;\ s/c = \{\cdots \pi_i := e_i, \cdots\}\}}$$

The typing rule for a generator is then:

$$\frac{[a : \text{type}],\ \rho \vdash_{\text{Coalg}} A(a)\{c := \tau;\ s/c = \{\cdots \pi_i := e_i, \cdots\}\}}{[a : \text{type}],\ \rho \vdash \mathit{gen}[s]\, A(a)\{c := \tau;\ s/c = \{\cdots \pi_i := e_i, \cdots\}\}\ :\ \tau \rightarrow s(a)}$$

To give a typing for *cohom*, one must type a splitting function in addition to a coalgebra specification.

$$\frac{\begin{array}{l} g\ :\ (\tau_1 \times \cdots \times \tau_n) \rightarrow \tau' \\ [a : \text{type}],\ \rho \vdash_{\text{Coalg}} A(a)\{c := \tau;\ s/c = \{\cdots \pi_i := e_i, \cdots\}\} \end{array}}{\rho \vdash \mathit{cohom}[s]\, A(a)\{c := \tau;\ s/c = \{\cdots \pi_i := e_i, \cdots\}\}\ g\ :\ \tau \rightarrow \tau'}$$

7.3 Termination conditions for *cohom*

A function defined by a *cohom* combinator selects a particular path for access of a value from its coalgebra. Thus any property that can be inferred of all finite paths in the coalgebra will hold for any path selected by an application of a function defined by a *cohom*, provided that the path is finite. The additional proof obligation for a *cohom* is just a proof of finiteness, or a termination proof.

A finiteness proof can be formalized as a proof that the set of paths that may be selected by a *cohom* is well-ordered. Typically, such a proof will hold only when the domain of the function is restricted by a predicate. A finiteness predicate can be inductively defined through clauses that mimic the structure of a coinductive proof.

An inductive proof consists of a set of implication schemas and a rule establishing that a given predicate holds of each element of a set, by a finite chain of implications drawn from instances of the schemas. Inductive proof is an argument by which to establish a property of a set in terms of its construction. The construction of morphisms of certain varieties of structure algebras help to shape the necessary finiteness arguments in terms of well-ordering relations, thus the technique is applicable to establish properties of the codomains of algebra morphisms of these varieties.

Dually, A coinductive proof consists of a set of implication schemas and a rule establishing that a given predicate holds for each witness of a set, by a finite chain of implications drawn from instances of the schema. Coinductive proof is an argument by which to establish a property of a set in terms of its witnesses. The construction of morphisms of certain varieties of coalgebras helps to shape the finiteness arguments in terms of well-orderings, thus the technique is applicable to establish properties of the domains of coalgebra morphisms of these varieties.

A finiteness predicate is the least specific assertion that can be made about the domain of a coalgebra morphism. It asserts only that all finite witnesses are defined. Thus a finiteness predicate characterizes the domain of a coalgebra morphism. The structure of such a finiteness proof is that

- a predicate, P, holds of some initial values by direct implication from facts, i.e. without use of any implication that involves P in its hypothesis, and

- for every projector, π_i of the coalgebra, let $(x_1, \ldots, x_n) = \pi_i\, x$.
  Then if c occurs in the typing of π_i, i.e. $\exists j \in 1..m_i \mid \sigma_{i,j} = c$,
  $$\left(\bigwedge_{\{j \mid \sigma_{i,j} = c\}} P(x_{i,j}) \right) \Rightarrow P(x)$$

Example 7.4 :
For the construct *while*(p, r), a finiteness predicate is the least specific that satisfies:

$$\begin{aligned} p\,x = ff &\Rightarrow P(x) \\ P(r\,x) &\Rightarrow P(x) \end{aligned}$$

The implications indduce a well-ordering relation $(\prec)$ on the set characterized by P, namely that $\forall x.\ P(x) \wedge (p\,x = tt) \Rightarrow r\,x \prec x$. Thus $P(x)$ is a termination condition for the evaluation of *while*$(p, r)\, x$.
□

8 Summary

ADL provides a great deal of detailed structure to help a programmer to formulate well-understood, readily verified algorithms. Its structure is based upon the idea that computation is essentially algebraic. Thus ADL employs structure algebras and coalgebras as the fundamental concepts for building algorithms. By comparison with a more traditional functional programming language which uses recursion as the basic control mechanism in defining functions, ADL has a richer type system, a more explicit verification logic, and provides a finer granularity of control over evaluation. Well-typed ADL programs are well-behaved in the sense that every computation either terminates or else it delivers finite output in response to finite input. There are no divergent computations.

Basing a programming language upon well-defined mathematical structures has certain definite advantages:

- The programming language has a rigorous theory that can be formalized in a logic. Properties of programs can be established directly by formal reasoning about ADL program fragments, in the ADL programming logic. Verification does not depend upon the indirect step of reasoning about the semantics of the programming notation.

- Many aspects of the meaning of a program can be carried in the type system of the language. The type system is a specialized abbreviation of the programming logic, one that is often easier to comprehend and to validate than is the full verification logic. Typing information is also useful for the implementation of ADL.

- Programmers benefit by having more, rather than less structure for their designs. Structure, if understood, helps to decompose problem solutions with a useful modularity. The algebraic structure of ADL, while not immediately familiar to programmers, is easily learned and understood.

The ADL language has been designed to lend itself to transformational development, i.e. the systematic improvement of algorithms by meaning-preserving, algebraic transformation of programs. Transformational development is an old idea, but the algebraic aspect of program transformation has been emphasized by Richard Bird [Bir84, Bir91] and his coworkers. The deforestation algorithms proposed by Wadler [Wad88] furnish a good example of general transformations. Wadler [Wad89] and Malcolm [Mal89] have observed that there are general classes of theorems that have instances for any inductive datatype. Such theorems are not only useful in justifying transformations, they may be automated as tactics for the application of term rewrites that actually effect the transformations. This observation is the basis for a higher-order transformation tool (HOT) currently being developed for use with ADL programs.

The initial applications for ADL have been as an executable semantics specification language. ADL has been used to program a computational semantics for a domain-specific design language. It can be used for many other applications, however. It should be particularly advantageous for applications for which formal verification is desired.

A Syntax

The syntax $foo^{\{,\}*}$ means *foo* may be repeated zero or more times, with a , in between each instance. The $^{+}$ is similar to *, but the item may be repeated *one* or more times, and $^{++}$ indicates *two* or more times. The angle brackets ⟨ ⟩ indicate optional items.

A.1 Base syntax

The non-literal terminals are *id* (identifiers) and *const* (special constants such as integers and strings).

$decl$	::=	$\mathtt{signature}\ id\ \langle\mathtt{(}id^{\{,\}+}\mathtt{)}\rangle\mathtt{\{}\ tvars\ \mathtt{;}\ sortsig^{+}\ \mathtt{\}}$
		$\mathtt{cosignature}\ id\ \langle\mathtt{(}id^{\{,\}+}\mathtt{)}\rangle\mathtt{\{}\ tvars\ \mathtt{;}\ cosortsig^{+}\ \mathtt{\}}$
		$\mathtt{val}\ valbind$
		$\mathtt{prefix}\ int\ id^{+}$
		$\mathtt{infix\ left}\ int\ id^{+}$
		$\mathtt{infix\ right}\ int\ id^{+}$
$tvars$	::=	$\mathtt{type}\ id^{\{,\}+}$
$sortsig$	::=	$id_{sort}\ \mathtt{/}\,id_{carrier}\ \mathtt{=}\ \{\ opdecl^{\{,\}+}\ \}$
$opdecl$	::=	id
		$id\ \mathtt{of}\ type$
$cosortsig$	::=	$id_{sort}\ \mathtt{/}\,id_{carrier}\ \mathtt{=}\ \{\ coopdecl^{\{,\}+}\ \}$
$coopdecl$	::=	$id\ \mathtt{:}\ type$
$type$	::=	$id\ \langle\mathtt{(}type^{\{,\}+}\mathtt{)}\rangle$
		$type^{\{*\}++}$
		$type_1\ \mathtt{->}\ type_2$
		$\mathtt{(}\ type\ \mathtt{)}$
$valbind$	::=	$pat\ \mathtt{=}\ expr$
pat	::=	$apat$
		$id\ \mathtt{as}\ pat$

apat	::=	_
		id
		($pat^{\{,\}*}$)
rulepat	::=	_
		id
		const
		id pat
		id **as** *pat*
		($pat^{\{,\}*}$)
expr	::=	*aexpr*
		appl
		apat expr
		let *valbind* **in** *expr*
appl	::=	*expr expr*
		expr id expr
aexpr	::=	*id*
		$*id*
		const
		map[*id***]**
		red[*id***]** *algebra*
		hom[*id***]** *algebra*
		gen[*id***]** *algebra*
		cohom[*id***]** *algebra*
		case *expr* **of** $rule^{\{\mid\}+}$ **end**
		($expr^{\{,\}*}$)
rule	::=	*rulepat* **=>** *expr*
algebra	::=	$\langle id\langle(id_{typaram}{}^{\{,\}+})\rangle\rangle\{$ *tybind* ; $opbind^{\{,\}+}$ }
		$\langle id\langle(id_{typaram}{}^{\{,\}+})\rangle\rangle\{$ $tybind^{\{,\}+}$; $subalg^{+}$ }
tybind	::=	*id* = *type*
subalg	::=	*id*{ $opbind^{\{,\}+}$ }
opbind	::=	*id* := *expr*

Notes:

- In *type*, `*` binds more tightly than `->`, and `->` associates to the right.

- In *sortsig*, $id_{carrier}$ must be declared in *tvars*, and each $id_{carrier}$ may only appear in one *sortsig*.

- The ambiguity of application in patterns and expressions is resolved by a precedence parser. Each *id* has an associated fixity (prefix or infix), precedence and, for infix, associativity. Precedence is a positive integer, with higher value indicating higher precedence (binds more tightly). These are assigned by a `prefix` or `infix` declaration. The default is prefix with precedence 9.

- No *id* may appear twice in a *pat* and no *id* may be that of a free algebra operator. In a *rulepat*, no *id* may appear twice, unless it is a free algebra operator.

- In an *algebra*, the *ids* bound in *tybinds* must correspond to the *tvars* declared in the algebra signature.

A.2 Derived forms

$id\ apat_1 \ldots apat_n$ `=` $expr$	$\Rightarrow$	id `=` `\`$apat_1 \ldots$ `\`$apat_n\ expr$
`let` $valbind^{\{\texttt{and}\}++}$ `in` $expr$	$\Rightarrow$	`let` $valbind^{\{\texttt{in let}\}++}$ `in` $expr$
`if` $expr_1$ `then` $expr_2$ `else` $expr_3$	$\Rightarrow$	`case` $expr_1$ `of true =>` $expr_2$ `| false =>` $expr_3$

References

[Bir84] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984. Addendum: Ibid. 7(3):490-492 (1985).

[Bir86] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO Series F*. Springer-Verlag, 1986.

[Bir88] Richard S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 52 of *NATO Series F*. Springer-Verlag, 1988.

[Bir91] Richard S. Bird. Knuth's problem revisited. In B. Möller, editor, *Constructing Programs from Specifications.* North-Holland, 1991.

[CS92] J. R. B. Cockett and D. Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory, 1991.* AMS, 1992.

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Twente, The Netherlands, February 1992.

[Gog80] Joe A. Goguen. How to prove inductive hypotheses without induction. In W. Bibel and R. Kowalski, editors, *Proc. 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 356–373. Springer Verlag, 1980.

[Hag87] T. Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[Mal89] Grant Malcolm. Homomorphisms and promotability. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, June 1989.

[Mee86] Lambert Meertens. Algorithmics—towards programming as a mathematical activity. In *Proc. of the CWI Symbposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[Pau93] Lawrence C. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report TR 304, Computing Laboratory, Cambridge University, December 1993.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer-Verlag, March 1988.

[Wad89] Philip Wadler. Theorems for free! In *Proc. of 4th ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989.

Volume VI
Results of the SDRR Validation Experiment

Results of the SDRR Validation Experiment

Richard B. Kieburtz

February 27, 1995

Abstract

This report presents the principal results of an experiment in software engineering in which the SDRR technology, a new method for constructing program generators, was applied to an often-repeated application domain-- message translation and validation. A generator designed and implemented with SDRR was compared with an existing state-of-the-art solution for the same problem domain, based upon a set of reusable Ada program templates.

The experiment employed four subjects to conduct trials of use of the two technologies on a common set of test examples. The experiment was conducted by an independent contractor. Test examples consisted of Air Force message specifications taken from C^3I systems.

The results show that the subjects achieved significantly greater productivity and introduced fewer errors when using the program generator than when using the templates to implement software modules from sets of specifications. These difference shown in these results is statistically significant at confidence levels exceeding 99 per cent.

1 Introduction

An experiment was planned and conducted to evaluate an application of the SDRR method [1] on an Air Force software application domain. The application domain that was selected was message translation and validation (MTV) for C^3I systems [2,3]. An MTV module is required for the message format of each sensor in a C^3I system. Its function is to translate incoming messages that arrive as streams of bits or bytes into an internal data structure that can be further interpreted by a controller. As a message is translated, validity checks may be performed on fields of the message. The MTV application is a natural one to which to apply software generator technology, as a typical C^3I system will incorporate several different MTV modules to translate messages from its several sensors. Each of these modules has similar requirements but differs in a multitude of details. Thus conventional reuse of software components fails with MTV modules.

PRISM (portable, reusable, integrated software modules) was chosen as a host architecture for MTV modules that were to be produced by a software component generator built with the SDRR technology. A factor in the choice of PRISM was the existence of a software development system for MTV modules using a state-of-the-art technique to achieve design reuse[4]. This is the MTV Model Solution, developed for the Air Force by the Software Engineering Institute. The Model Solution provides a set of 12 Ada program templates that are customized to produce a specific MTV module. We wanted to compare the SDRR technology with the best available state-of-the-art solution for software component development, and the Model Solution represents this.

An MTV module for the PRISM architecture consists of an Ada package containing six functions. The first function translates the external, bit-string representation of an arriving message into its internal representation as a value in an Ada data type; the second function translates the internal representation into a printable ASCII string, the user representation, that can be logged or printed. The third and fourth functions invert the translations of the first two, going from user to internal representation and from internal to external. The last two functions perform validity checks on messages that arrive in external representation or are entered in user representation.

The MTV Model Solution synthesizes an Ada package from a set of edited templates. The MTV Generator (MTV-G) generates an Ada package directly from a specification of the desired message format, written in declarative form in a message specification language (MSL) that was designed by OGI for this application as a part of the development of MTV-G. In a 12-week long experiment, the performance of a group of impartial subjects in using the templates-based Model Solution was compared with their performance using the MTV Generator to carry out a sequence of development tasks. These tasks involved both the design and modification of MTV modules for a variety of message specifications. This report summarizes the results of that experiment.

2 Hypotheses of the Experiment

The experiment was designed to test several hypotheses comparing the MTV Generator that was developed with SDRR technology with the state-of-the-art MTV Model Solution that is based upon reusable Ada templates. The Ada templates technology exemplified in the Model Solution has already established that it enhances productivity and reduces defect insertion rates when compared with custom coding techniques. The specific hypotheses that were tested in this experiment are:

Productivity - *Hypothesis:* Developers will be more productive in creating or modifying a message translation and validation module when using the MTV-G technology than with the templates-based technology.

Reliability - *Hypothesis:* Fewer defects will be introduced in development with MTV-G than in a solution developed with templates.

Flexibility - *Hypothesis:* The MTV Generator can be configured to at least as many instances of message specification as can the templates.

Predictability - *Hypothesis:* More accurate predictions of the cost and risk of creating or changing an MTV module will be made if the development is done with the generator technology than with templates.

Usability - *Hypothesis:* Developers will find the generator technology easier to use than the templates.

3 Experiment Design

Four subjects were engaged in parallel software development and maintenance activities to compare the two different technologies for producing software components. The subjects were trained in the use of both technologies, including hands-on experience with sample problems prior to beginning the experiment.

3.1 Subjects

Intermetrics, Inc. contracted to provide and supervise four subjects to conduct trials for the experiment. These persons were selected to have similar qualifications and experience. Each held a BS or MS degree in computer science and had one to three years Ada programming experience. There were two males and two females. They had no prior knowledge of the application domain. These subjects are probably somewhat overqualified for the particular tasks required of them in conducting the experiment, but it was felt that Ada experience was required in order not to put them at a disadvantage in using the Model Solution which is based upon a set of Ada program templates. And indeed, they were able to resolve technical problems with the templates that arose in the course of the experiment. These problems would probably have stymied less qualified subjects.

The experiment trials were conducted at the Intermetrics facility in Cambridge, Massachusetts. Direct supervision of the subjects was provided by Intermetrics management.

3.2 Training

Training in use of the MTV Generator and the Model Solution templates was provided during a ten-day period prior to the start of the experiment trials. The subjects also received training in use of the experiment environment in which they were to conduct their work and in the procedures they were to follow during the course of the experiment. The trainer for MTV-G was Jef Bell of OGI, a technical staff member of the Pacific Software Research Center. The trainer for the Model Solution was Chuck Plinta of Accel Software Engineering. Mr. Plinta was one of the developers of the Model Solution at the Software Engineering Institute of Carnegie-Mellon University. Accel is commercializing the Ada templates technology.

The training in each technology was completed in three days, plus an additional two days of practice time. Two of the subjects were trained first in MTV-G and second in the Model Solution. The order of training was reversed for the other two subjects to eliminate order of training as a possible source of systematic bias. Following training, all subjects participated in a series of mock trials for three days to consolidate their training and gain familiarity with the software tools provided in the experiment environment. Each subject was assigned an equal number of tasks in each technology during the mock trials, but some subjects had extra time and performed a few additional, unscheduled tasks in the mock trials.

3.3 Work Environment.

Each subject was furnished a computer workstation (Sun Sparc Classic running SUN/OS) on which to perform his/her work. The workstations had a common software complement including the emacs editor, RCS version archiving, the Sun (Verdix) Ada compiler, and other tools commonly found on a UNIX system. The RCS archive of each file that was edited was automatically updated at the conclusion of the edit session. This provided a detailed record of edit sessions. It logged the date and times at which an editing session was started and finished, the file that was edited, the number of lines that were entered and deleted, and it identified the subject who performed the editing.

Compilations, system builds and test runs were also controlled through a set of Tcl scripts that defined the work environment presented to the subjects. The environment recorded the times and outcomes of compiles, builds and test runs. All of the data that were automatically logged by the environment were recorded without intervention by the subjects.

Although the subjects were informed that the results of their work were being monitored, they received no feedback from the system as to what data were recorded or when the records were made or collected. Data collection was as unobtrusive as possible. Furthermore, the subjects were told in advance that the data collected in the experiment would not be used to evaluate their performance as individuals. The time sheets kept by the subjects were mailed directly to OGI and were not given to Intermetrics management In fact, after the experiment trials were concluded, one subject requested a performance review from OGI. This request was denied on the grounds that there had been prior agreement not to disclose data on the performance of individuals.

4 Flow of work

At their training session, the subjects were given written instructions on the conduct of the experiment (see Appendix A). Each subject also signed a participant consent form, which informed him/her on the uses to be made of the data that were to be collected (Appendix B).

4.1 Task initiation

The task schedule for the experiment was prepared at OGI in advance of the experiment. The ICD specification for each task was sealed in an envelope that had the task identifier written on the outside. These envelopes were sent to the Intermetrics supervisor of the experiment. The supervisor maintained a spreadsheet record of the assignment and completion status of each task (Appendix C).

When a subject was ready to start a new task, he/she obtained the task specification from the experiment supervisor at Intermetrics. The first action taken by the subject was to checked in the task identifier with the environment, to log the start of a new experiment trial. This action allowed the subject access to files pertinent to the assigned task. Similarly, when a task was completed, the subject logged this event. The environment would not permit a subject to have more than one task active at any time. Subjects were requested to exit from the environment to perform non-experiment-related personal tasks, such as reading e-mail, so that time would not be accounted for these.

Subjects were asked not to work on experiment tasks outside the environment. In post-experiment interviews, the subjects were asked whether they had conformed to this work rule. Each subject indicated that the per cent of time they had spent working outside the environment was small (estimated at 5-10%) and was spent in solving unusual problems. These included tracking down an Ada compiler problem, confirming a suspected problem with a set of acceptance test data and writing and testing a new Ada template needed to extend the Model Solution to an out-of-scope design task. These problems were beyond the scope of activities supported by the programmed experiment environment.

The subjects were also told not to work collaboratively on tasks and never to show one another the ICD specifying the task on which they were currently working. They were allowed to ask one another technical questions, and did so. In the post-experiment interviews, each subject was asked about interaction with the other subjects. All reported that these interactions were of short duration, and predominately concerned general problems, such as the Ada compiler bug, problems with data or the environment, and work-arounds.

The designated supervisor for the experiment at Intermetrics monitored overall progress of each subject in completing assigned tasks. A weekly status report was sent to OGI (see Appendix D). Subjects were allowed some flexibility in scheduling their working hours, but were asked to confine their workday to hours between 8am and 8pm. This provided a 12-hour period for remote data collection without risk of conflict with subject activity.

4.1.1 Time and effort prediction

When a subject checked in the start of each new task with the environment, he/she was required to estimate the time of completion and the total effort hours that would be expended to perform the task. The accuracy of subject estimates was a factor that was to be measured to see if there was a difference that could be attributed to the technologies.

4.2 Technical support

Technical support for problems or questions raised by the subjects was provided by OGI via telephone and e-mail. Jef Bell and Laura McKinney, the SDRR project manager, served as the experiment monitors and primary points of contact at OGI for the experiment subjects. For technical problems concerning the Model Solution templates, subjects were allowed to contact Chuck Plinta by telephone. A summary of the procedures used by the OGI monitors is in Appendix E.

When a subject was blocked on completion of a task awaiting a response to a technical question, the work environment permitted the current task state to be changed to "suspended" so that a new task could be opened and time would not be lost. To resume a suspended task, the subject had to complete or suspend any other task that might have been taken up. This discipline was enforced by the environment.

4.3 Task completion criteria

The normal criterion for completion of a task was that the code generated for the message specification passed an acceptance test. Acceptance test data were automatically generated at OGI for each of the experiment tasks, and were loaded into the experiment environment in advance of the trials. An acceptance test suite comprised 25 acceptable messages and 25 that violated the specification. Special attention was given in generating the test data to ensure coverage of field value constraints when these arose in a message format specification. However, there were a few cases in which test coverage of all constraints was not achieved.

The experiment environment provided a test harness that could be invoked by a subject to test a module that had been created. Test runs were identified as either unit testing or acceptance testing. A subject provided his/her own test data for unit testing and the outcomes were not recorded. For acceptance testing, the test data used were those that had been pre-loaded into the environment. Outcomes were recorded. Completion of a task occurred when it passed an acceptance test.

The prototyping capability of the MTV Generator allowed full-function testing at the prototype level. Thus a subject could complete acceptance testing of an MTV-G module prior to actually generating Ada code. Because of the time needed to run the program translation tools and ultimately build and compile an Ada package, the subject was allowed to initiate these runs off-line, during lunch breaks or overnight. The Ada package was then subjected to confirmation tests against the same acceptance test data. Successful completion of these tests terminated the task.

Because the MTV Model Solution is comprised of Ada templates, it could only be tested after building and compiling an Ada package. The same acceptance test data were used to validate the completion of a common task in either technology.

In the few instances in which a solution could not be constructed due to Ada compiler problems, a task was completed abnormally without having passed acceptance tests or without extending the acceptance tests beyond the prototype stage, if the task was performed with MTV-G.

4.4 Trial task specifications

Message formats were specified by an Interface Control Document (ICD), a tabular form that specifies a message layout field-by-field. It names each field, prescribes its length or its terminator if it is of variable length, and gives the intended interpretation. The ICD's for use in the experiment were reviewed prior to the beginning of trials by the OGI project manager, who obtained clarification on points of ambiguity or apparent inconsistency from the originator of the trial specifications at ESC. After review, the specifications were given to other OGI personnel who were not otherwise involved in the design or analysis of the experiment, to generate acceptance test data for the specification. These test data were then reviewed for accuracy and coverage, and subsequently installed in the experiment environment. The review process is described in Appendix F.

In spite of the reviews of acceptance test data, there were several instances in which subjects discovered problems with test data during the course of the experiment. The causes included a version control problem, a misinterpretation of the intended meaning of a field in an ICD, and an instance in which a range constraint on a data field had been ignored. Each of the problems was corrected, allowing the affected task to be completed.

Message format specifications for the experiment trials were furnished by ESC personnel, who assembled these from actual, unclassified Air Force message specifications taken from a number of different C^3I systems. These specifications are quite rich in detail. The number of field types in an individual message varied from 7 to approximately 40. Both ASCII character-based and bit-based fields were included. Variable-length fields, optional fields, lists of fields and variant record fields all occurred. Up to 232 alternative string patterns were specified in a single variant field. Some fields specified value ranges and numeric scaling. Inter-field value constraints were encountered. An example ICD is given in Appendix G.

4.5 Experiment trial sequences

The experiment was designed to measure performance of the subjects in producing software packages for translation and validation of messages in a wide variety of format specifications. In addition, the experiment measured performance in maintaining modules as they underwent a cumulative series of changes in the message format specification.

The original specifications consisted of 12 independent message formats from which instances of the translation and validation module were designed. These were followed by 56 specified modifications. Eight of the original message format specifications were chosen for simulated maintenance. The maintenance tasks formed eight series, each consisting of seven cumulative modifications to the original ICD. Each of the message specifications was implemented in each of the two technologies, thus there were 24 original design tasks and 112 modification tasks to be assigned among the four subjects.

Task identifiers consisted of a major task number, its modification number, and the technology in which it was to be implemented. For example, the string "6.0.msl" designated the task of implementing original ICD number 6 with the MTV-G technology.

Each subject implemented six of the original ICD's, three in each technology. Each subject then implemented 28 modification tasks, randomly selected except that 14 were to use MTV-G and 14 were to use the Model Solution templates. The task sequences that were assigned to individual subjects were randomized, subject to the constraint that a subject alternated use of the two technologies in the initial six tasks. No subject was assigned to implement the same original message specification in both technologies. The first task assigned to a subject required use of the same technology in which the subject was trained first.

4.5.1 Out-of-order task performance

Although the modification tasks were assigned to subjects in random order, subject to the constraint that task technologies were programmed to alternate for each subject, these tasks were also constrained by the sequence of modifications. A particular modification task, say 6.4.msl, could not be performed until its predecessor, 6.3.msl, had been completed. Since subjects worked at different rates, it frequently occurred that a subject was blocked on his/her next assignment pending completion of a predecessor task by another subject. In order to maintain work flow in the experiment, a subject whose assigned task was blocked awaiting completion of a predecessor was allowed to progress to his/her next assigned task and perform it out of order. There were many cases in which this occurred.

At one point, the majority of subjects became blocked on all future tasks. To keep the experiment running, a decision was taken to start a second thread of task activity, to be carried on while subjects were blocked on the main thread. For the second thread, the ICD's were renumbered with major task numbers from 13 through 24 and were assigned to the subjects whose work on the main thread was blocked. Subjects were not assigned to perform the same task in the second thread that they had been assigned in the main thread. In one instance, a task completed in the second thread had been specified from the same ICD and technology as a task that remained incomplete in the primary thread. In this case, the completed design artifact was allowed to be substituted for the incomplete solution in the first thread, which unblocked a task sequence so that subjects could return to tasks in the first thread. Although data were gathered for performance of tasks in the second thread, those data have not been analyzed and do not form part of the basis for the conclusions drawn in the experiment.

4.6 Out-of-scope tasks

Since the data for the experiment trials were assembled from a variety of existing Air Force ICD's, it was anticipated that some message specifications would be out of scope for the design of one or the other MTV solution. Subjects were instructed to report out-of-scope problems to their technical support contact and suspend the task until they received further instruction. For tasks to be done with the MTV-Generator, the intended remedy was to supply the subject with an extended version of the MSL compiler that provided features that would enable the task to be completed. For tasks to be done with the Model Solution, the intended remedy was that new templates would be written, either by the subject or by the technical support person, so that the task could be completed. However, no attempt was made to provide or suggest such extensions to the basic technology unless they were specifically requested by a subject.

As it turned out, fewer MTV-G tasks were deemed to require out-of-scope extensions to than had been anticipated by the OGI research team when they inspected the task specifications. The subjects were able to find work-arounds to complete most tasks without requesting modifications. In one case, the subjects did request and receive an extension to the MSL language to allow a task to be completed.

Several tasks required the subject to modify templates in order to complete the tasks in the Model Solution. The subjects were generally successful in making the necessary modifications to the Ada templates. In one case, a subject requested guidance in writing an entirely new template for processing variable-length character strings for the Model Solution and was able to accomplish this without unusual difficulty. Our conclusion is that both technologies demonstrated the capacity for extension to meet new requirements, given the availability of technical expertise to implement the needed changes.

5 Data collection

The experiment produced a large amount of data in several forms. The data that were directly reported or were extracted from computer files maintained by the experiment environment consist of:

- Task effort allocation reports, submitted by the subjects on a weekly basis. These reports attribute effort hours to tasks. A copy of the reporting form, together with the instructions given to subjects about reporting time, are contained in Appendix H.
- Task summary reports. These include start and completion times for tasks, the identifier of the subject who performed the task, and the subject's estimate of total elapsed time and hours of effort that would be required to complete the task. These reports were extracted from the log file generated by the experiment environment.
- Task assessment forms. A task assessment form was filled out by a subject as he/she completed each task, or at the end of a week if a task was still being attempted. It asks a series of questions that can be answered with a number on a scale of 1-5 to assess the subject's perception of the difficulty of the task and the usability of the technology and environment. It also asks for a short statement of any problems that were encountered in performing the task. A copy of the task assessment form is contained in Appendix I.
- Session summaries logged by the experiment environment. These record the subject's identifier, the times at which the session began and ended, the times at which activities occurred (edit transactions, compilations, builds, test runs, etc.) the type of each activity and its outcome, if relevant, and the files accessed by the activity. The session summaries provide a detailed account of on-line work flow.
- Edit summaries extracted from RCS logs. The experiment environment was programmed to save file images to the RCS archive at the conclusion of each edit session. At each save, the archive records time, file name and the difference

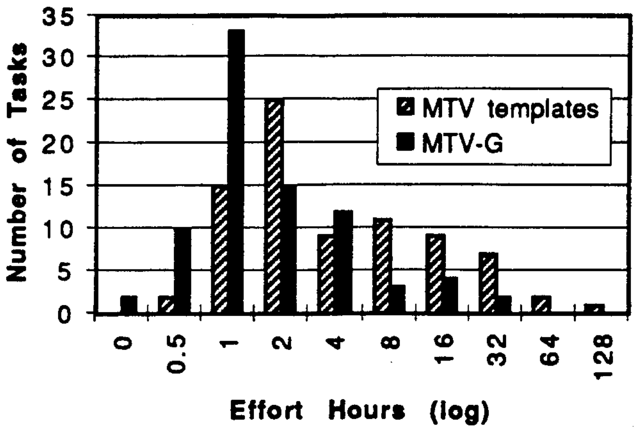


**Figure 2: Distribution function-
Effort on tasks**

Another measure of the subjects working time confirms the results obtained from the reported effort hours. The mean of accumulated time spent in editing, per task, was 2.83 times longer for tasks performed with the Model Solution.

The hypothesis of increased productivity by using the MTV Generator has certainly been confirmed. In evaluating the implications of these productivity data, it is important to consider what they do and do not include. It is conventional wisdom that there is little to be gained by improving the productivity of software engineers in coding alone, as that activity typically accounts for only 7 to 15 per cent of the total cost in a project.

The activity monitored and accounted for in this experiment accounts for far more than just coding. It encompasses conversion of application requirements (the ICD) into a specification, plus design, coding and unit testing of a solution. It does not account for original domain analysis (which is reused), nor for documentation and integration of the generated software artifact. For the family of frequently-repeated software modules that is represented by the MTV domain, the dramatic gain in productivity shown by the experiment is real.

6.3 Predictability of required effort

The predictions of effort hours that would be required to complete a task were correlated with the effort hours reported by subjects. For tasks done with MTV-G the correlation coefficient was 0.72. For tasks performed with the Model Solution, the correlation of prediction with actual effort was slightly lower, 0.63. It is not clear whether or not this difference is significant.

Another interesting statistic is the distribution of the ratio of reported effort to that predicted. One might expect this distribution to be normally distributed around a mean of 1.0, but this is not the case, as can be seen from the chart in Figure 3.

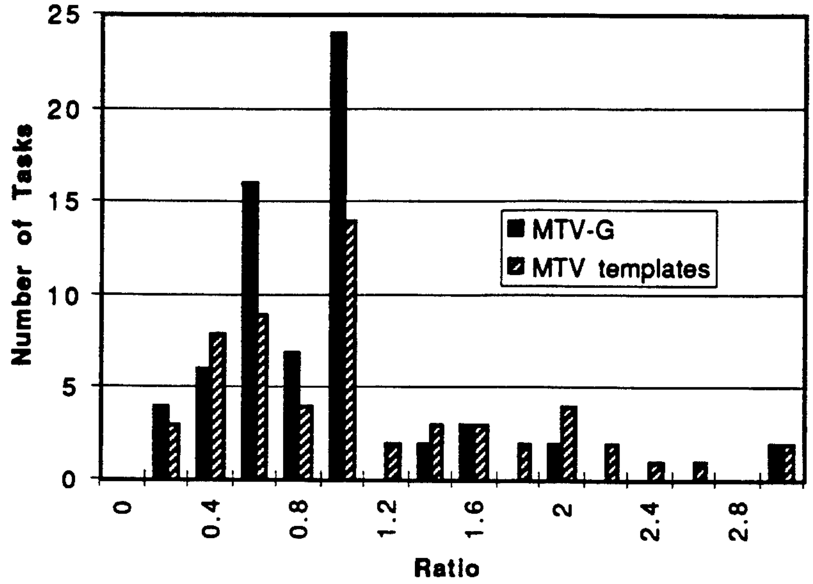


Figure 3: Distribution of ratio of reported to predicted effort hours

In this graph we see that the effort required is substantially overestimated, particularly for tasks to be performed with MTV-G. As mentioned previously, there is a bias towards overestimation of modification tasks which tended to be very easy. But the distribution is so skewed that the bias introduced by the granularity of time estimates cannot explain the whole effect. Apparently the accumulated experience of the subjects misled them in estimating the time required to perform tasks with MTV-G.

The outlying points toward the right of the graph are of particular interest, for they indicate cases in which a subject badly underestimated the effort that would be needed to complete a task. There are somewhat more of these points for tasks done with the Model Solution than with MTV-G. We do not know if this difference has statistical significance. The data are not inconsistent with the hypothesis that better predictability of effort is achieved with MTV-G.

6.4 Reliability

As a predictive indicator of the reliability of an MTV module operating in a system, we have measured the number of acceptance tests that a module has failed to pass prior to completion of the task. The instructions given to the subjects were that they could perform as much of their own testing of a module as they believed was necessary, and should not submit it for acceptance testing until they were confident that it was correct. If a module passed its acceptance test on the first submission, its failure score was zero. Each additional time that an acceptance test was run on the module added one to its failure score. Work on a module was not completed until it passed an acceptance test run. The same suite of test data was applied to the module on each run.

Figure 4 shows the distribution of acceptance test failures for tasks performed in each technology. The graphs appear to approximate exponential distributions.

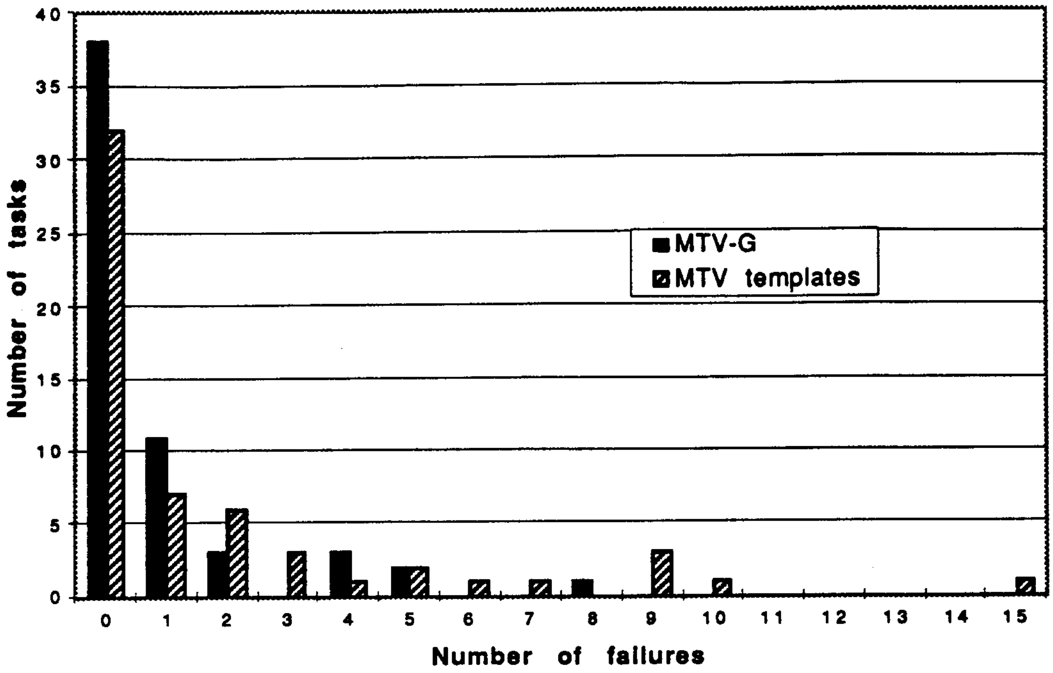


Figure 4: Distribution of acceptance test failu

The mean number of failures for MTV-G tasks is 0.8; for Model Solution tasks it is 1.8. These data were subjected to an analysis of variance, which confirms the significance of this difference with a 97% confidence level. This factor of nearly 2.3 in the mean number of failed test runs confirms the hypothesis that fewer defects are introduced in modules implemented with MTV-G.

6.5 Usability

After completion of the experiment, individual interviews were held with each subject. The subject was asked to respond to a series of questions that had been prepared in advance (see Appendix J). The interviews were taped, and transcripts of the subject's answers were prepared for analysis. The questions were designed to evaluate a number of factors that might have contributed to or detracted from the subject's performance. Thirteen questions explored various topics concerning the usability of the two technologies that were compared. The most relevant questions, and quotations from the subjects' answers are given below: The subjects referred to the MTV Generator as MSL, which is the acronym for the message specification language that is its interface. Some of them referred to the Model Solution as TPL, which was the identifier appended to task identifiers that were performed in this technology.

Q. In which method did you have most confidence in the correctness of your solution?

"The MSL solution. If the MSL failed at compile time, you could easily find the errors. In the Templates solution, there could be errors that would not be known until runtime."
"The MSL solutions, especially because of the difficulties with out-of-scope tasks in the model solution."
"I was confident in both methods.... I had most confidence in the Template solution because you can immediately see Ada code."

"I had the most confidence in MSL."

Q. During the cycle of ICD modifications, which technology provided implementations that were easy to change?

"MSL was easier for sure. You had only one file as opposed to TPL."
"The MSL solution is easy to maintain. Its advantage is that everything is in one place."
"MSL was generally easier. For TPL, you might have to re-instantiate the template. Sometimes lots of changing was necessary."
"In the MSL solution it is easier to make changes than in the template solution."

Q. Please respond for each technology individually: What made this method easy or difficult to use?

"In the Template solution, ... I am not an extremely experienced Ada programmer.... The frustrating thing about MSL is the amount of time it takes to compile. ... If something did not work, it is easier to fix it in MSL."
"For TPL, the search and replace was easy and having the debugger was helpful. Implementing variable length lists was possible but it required a lot of typing. It shouldn't be so hard.
For MSL, the fact that it was a high level language made it easy. However, the error messages were not good..."
"MSL was easy because everything is in a single file. TPL is difficult because it is fragmented."
"For the Template solution it is nice to be able to break things up by files...
For the MSL, it is sometimes good to have everything in one file. However, the error messages are worthless..."

Q. Which method would be easier for you to train somebody else to use?

"MSL. There really is not that much one needs to know."
"If they do not know Ada, then the MSL method is easier to learn."
"If the person to be trained has Ada knowledge, then the model solution is easier,..."
"The MSL technology would be easier to train ..."

Q. If you were to return to this project six months later to make maintenance changes, which implementation would you choose to work with, MSL or the Model Solution?

"I would choose MSL because the changes are easy."
"MSL modifications are easier, assuming you can remember MSL."
"Since I work with Ada, I would go with the Template Model because Ada is fresh in my mind."
"I would prefer to do maintenance in MSL."

Q. If you were required to use an implementation of an ICD from this experiment in a real system with expected maintenance, would you choose the use the Model Solution implementation or the MSL implementation?

"I prefer MSL for use in a real system."
"I would choose the Model Solution because MSL didn't succeed all the time."
"I would choose MSL if the build process were improved."
"The Model implementation because we can see the Ada code."

Q. Please respond for each technology individually: If the ICD were lost and you were required to reconstruct the ICD from the implementation, how difficult would this be?

"For the MSL implementation is should be relatively easy.... For the Template model, you would have to use the filenames as clues... This one would be more work."
"With MSL, it would not be that difficult to reconstruct an ICD. With TPL it would be more difficult."
"Reconstructing an ICD would be pretty easy from MSL except for alphabetic vs. string. From TPL, it would be a little harder. Out-of-scope issues causes things to be `hacked'..."
"In the MSL solution, for other than constraints, it would be pretty straightforward... In the Template solution you have to figure out how to put all the files together."

Q. In which technology is it easier to document a solution?

"We did not really look at documentation. The Template solution allows you to put a system overview into a separate file. MSL allows decent documentation comments within the file."
"The documentation process would be lengthier for the model solution because there are so many files."
"Both are the same."
"I never did commenting or documentation."

Q. In which technology will it be easier to maintain your skill level without retraining?

"Spending a half-hour per week doing a modification in each would be adequate to maintain efficiency in either technology."
"MSL.... it is concise, precise. Because of this, it was easier for me."
"If you had Ada knowledge, it is easier to maintain your skill level in TPL, but MSL is not hard to re-learn."
"...it depends upon what you are doing in the interim..."

Q. Please answer for each technology: How easy or difficult was it to trace the source of an error?

"In the MSL solution, 90% of the time it was pretty easy to trace the source of an error.... However, the error messages were at times incomprehensible.... In the Templates solution, a number of errors were related to the buggy Ada compiler. ... You needed to use the debugger to find these."
"With TPL, it is pretty easy if you know the (Ada) debugger.... With MSL it is more difficult because the error messages don't pinpoint the source of error."
"In TPL it was difficult ... In MSL, it was much easier."
"In the Template solution, it was easy to find an error based upon the error message. I did not like the MSL error handling...."

Q. Please answer for each technology individually: How easy was it to correct errors once located?

"It was easy regardless of which technology you are using. The big challenge was tracking down the error."
"Very easy."
"Easy in both technologies."
"With MSL, correcting errors was very easy. With TPL, it was usually pretty easy, if in scope. Except if fixing the error required you to re-type a lot of stuff. Changing early bit position was really awful."

Q. Which technology had the least opportunity for allowing error introduction?

"MSL worked better for me."
"With TPL, error introduction was mostly low.... With MSL it was also low, slightly better."
"The Template technology had the least opportunity for errors."
"There is less chance of introducing an error in the MSL solution. One problem with the Template solution pertained to deleting. It was too easy to delete an extra line."

Q. For each technology individually, please characterize the typical type of error made or problem discovered.

"In MSL, I did not have many problems. In TPL, I can't really say what a typical error was."
"For MSL, typical problems were mistyping the representation. Other problems involve the use of the curly brackets or the square brackets. Since they are on the same key, ... For the Template solution, you had to enter the length of things. The ICD's were not always accurate on lengths of items...."
"... most of my errors were spelling errors.... "
"With TPL, the typical type of error had to do with the ICD template (lengths or bit positions). With MSL, it was often cut and paste errors, or because you can't have subfields with the same name."

These comments indicate that (1) both technologies were acceptably easy to use, and (2) all subjects found MTV-G simpler to use than the Model Solution. (3) The strongest negative factors for MTV-G had to do with long compilation times and poor error messages; for the Model Solution they were the need to access multiple files and the larger amount of editing that was required to perform a task. Some subjects were more comfortable in being able to debug at the level of Ada, while others didn't miss it at all.

Interestingly, the subjects did not perceive a significant difference in the likelihood of error insertion between the two technologies, although the measurements of their work indicated that there was a difference.

Summary of results

This experiment has produced dramatic and definitive results demonstrating a pronounced advantage for the MTV Generator constructed with SDRR technology in the areas of productivity improvement and a lower rate of delivered defects from applications developers. These advantages have been realized in a direct comparison with the best currently available technology for developing software components. The following table summarizes the quantitative results of the analysis:

Productivity	**MTV-G**	**MTV Templates**	**Ratio**
Average effort hours per task	2.54	6.96	2.92
Reliability			
Average number of test runs failed	0.8	1.8	2.25
Predictability			
Correlation of effort	0.72	0.63	

predicted with effort
expended

Furthermore, the MTV Generator demonstrated outstanding robustness in its first trial -- the subjects encountered no errors in the MTV Generator. The most serious problem that they did encounter was its inability to handle large message specifications, a problem that has been substantially solved in a new version of MTV-G.

The results on predictability of effort that would be required to perform a task were inconclusive. Although the correlation of predictions with measured time was slightly higher for MTV-G than for the Model Solution, there was significant overestimation of the time needed to perform tasks with MTV-G, and with both technologies, there were outlying cases on which the predictions of effort were inaccurate.

The subjects favored MTV-G in their perceptions of its usability, although they had had no prior experience with it or any similar technology. Their biggest complaint concerning usability concerned the quality of diagnostic error messages, an aspect that was not emphasized in the implementation of MTV-G and obviously requires improvement.

The experiment has strongly confirmed the most important of the hypotheses about the advantages of generating software from specifications. It has demonstrated that SDRR is indeed a highly promising, new technology for improving the development of software components.

References

[1] Jeffrey Bell et al. Software design for reliability and reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.

[2] Richard B. Kieburtz. Software design for reliability and reuse—Method definition, February 1995. In [5].

[3] Jeffrey R. Lewis. A specification for an MTV generator. Technical Report 94-003, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[4] Charles Plinta, Kenneth Lee, and Michael Rissman. A model solution for C^3I message translation and validation. Technical Report CMU/SEI-89-TR-12 ESD-89-TR-20, Software Engineering Institute, Carnegie Mellon University, December 1989.

[5] Pacific Software Research Center. SDRR project Phase I final scientific and technical report, February 1995.

Appendix A Instructions for Subjects

The SDRR Proof-of-Concept Experiment

I. Experiment Environment

Purpose

As subjects in an experiment, you will be using a constrained and structured environment while performing work on the assigned tasks. Your work must be done in this prescribed manner to allow for statistically sound analyses of the results. The experiment environment should provide enough structure to control for unintended violations of the design, while allowing as much freedom as possible for you to do your work in the manner in which you are most comfortable.

Another concern in the design of the experiment environment is the protection of your personal privacy. All work you do will be tagged with a subject identifier rather than your name. You will be given a special system account to use while doing experiment tasks, which will be restricted to only allow access for you and the experiment monitors. We ask that you use personal accounts to do any other work or to handle electronic mail.

Naming Conventions

To facilitate communication and the use of the environment, we have developed several naming conventions for use in commenting, naming files, or referring to documents and tasks:

Subject Identifier: You will be identified to the system using a single letter identifier. For example, you might be subject *c*.

Technology Identifier: The technology in use for any particular task will be either *msl* (message specification language) or *tpl* (template method).

Interface Control Document Identifier: Each task will have an associated ICD. These will be numbered by original and modification numbers separated by a period (similar to version numbering for software). For example, the initial implementation of an ICD might be *5.0*, and sequential modifications to that ICD would be numbered *5.1, 5.2,* etc.

Task Identifier: Each task will be given a task identifier which indicates who should do the work, which particular ICD version should be implemented, and which technology is to be used. If subject *c* is working on the 4th modification to the 8th ICD in MSL, the task identifier will be *c.8.4.msl.*

File Naming: User-generated files associated with tasks will be prefixed using the ICD number of the specification it implements as well as the technology used in the implementation. For example, all user files associated with task *b.8.4.msl* will be named with the prefix *8.4.msl*, and followed by a user-defined name such as *weather*, which serves as a description of the file contents.

System generated files will be prefixed by the type of file, such as *results*, and suffixed with the ICD number and technology. For example, a file containing an inventory of all files used for this task might be named *inventory.8.4.msl.*

This convention allows the easy discrimination of user-generated and system files, as well as directly associating any file generated with the appropriate ICD specification it implements.

Files

Some files may contain automatically generated headers and comments. Please do not alter these in any way. You may use this history to discover who may have implemented previous changes to the file.

If you wish to maintain a personal RCS file for your current work, the system provides such a capability.

User-generated files for either MSL or templates will be maintained in a system RCS structure automatically. You will be able to check out files that directly pertain to your task.

Source Files

Source files will always have the appropriate prefix followed by a user-defined name. Source files are associated with specific tasks by inclusion in task directories. All source files will have a system-maintained comment header at the top that should not be altered.

Template Files

Template files are available to you through the environment. If you wish to use a template, a copy of the template file will be automatically generated and placed in your home directory. Template originals may not be modified.

Emacs

You are required to use the emacs editor in the experiment environment to do your work. The environment is constrained, and you may not work with multiple buffers.

If you have a personal .emacs file that you would like to use, please let the experiment monitor know so that may be arranged. Please do not alter the .emacs file yourself.

II. Task Description

Tasks

Your manager will have all the tasks you will be assigned during the experiment. Each task folder will be numbered to indicate the order that the tasks should be worked. Task folders will contain an ICD, with change bars if it is a modification, a Task Specification Form, and a Task Assessment Form.

Task Specification Form

A task specification form will accompany each ICD or change ICD. It will contain the task identifier and the technology to utilize. It will also include a time limit for task completion.

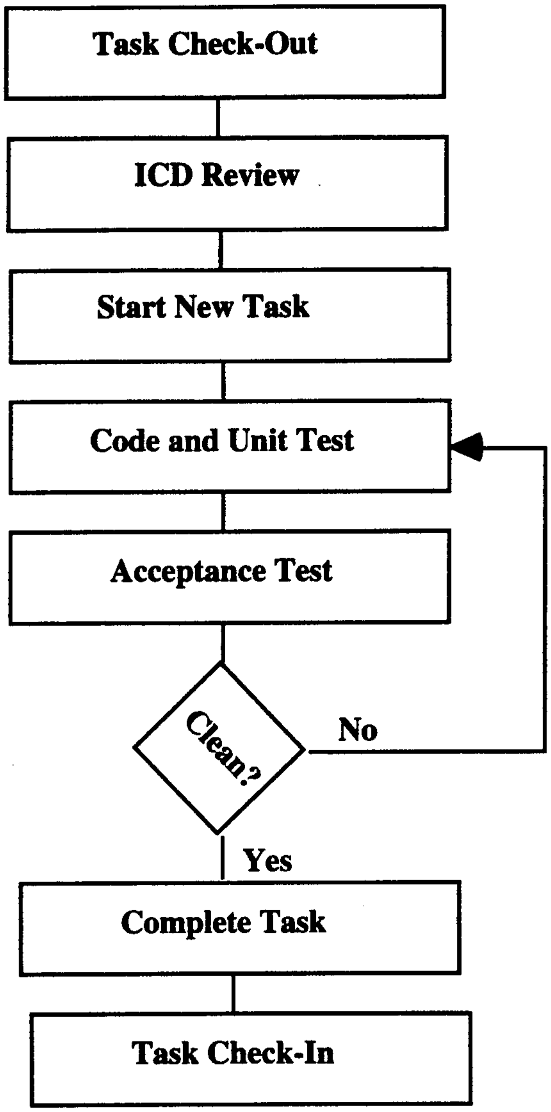
Task Assessment Form

A task assessment form will accompany each ICD or change ICD. It should be filled out after you have completed the task.

Questions or Problems with Tasks

If you have any questions regarding the contents of a task folder or the details of the ICD's or the forms, please contact the experiment monitor. If you discover any ambiguity or if the instructions are not clear, contact the experiment monitor. Your manager does not have any information about the contents of the tasks.

III. Task Process


Task Check-Out
ICD Review
Start New Task
Code and Unit Test
Acceptance Test
Clean?
No
Yes
Complete Task
Task Check-In

Task Check-Out

Your manager is responsible for assigning tasks. When you are ready for a new task, please see the manager. A chart will be maintained by your manager to indicate whether a task is blocked or not, and allow your manager to determine which should be the next task. Tasks should be checked out in the sequence order indicated, unless the task is blocked awaiting completion of a predecessor task by another subject. Tasks will not be checked out until previous tasks have been completed. When a task is checked out, your manager will record the check-out time and date on the chart.

ICD Review

You must do a careful review of the task specification, and generate estimates for completion. When you start the task on the system, you will be asked for these estimates before files will be checked out for work. Please give thoughtful estimates, as the experiment success depends on your assessments. Estimates will be given in two forms: 1) the estimated date of completion and 2) the estimated number of hours of effort. If you are planning to take personal time off or attend meetings, then the estimated date of completion may not correspond directly to the hours of effort that you estimate.

We are collecting estimates in order to assess the predictability of work in both technologies. Please do not adjust your pace of work to correspond to your estimate, but rather work at a regular steady rate.

Start New Task

When you have completed your ICD review, then you may login to the system using your special subject account. You should select the Start New Task option on the Main Menu. The system will ask you for your completion estimates. Then, if the task is a modification, all existing files will be placed in the task inventory.

Original ICD

For original ICD's, there will be no existing files.

Modification

For modifications, an automatic check-out of all the existing files will be done, and the files will be appropriately renamed and listed in the task inventory.

At this time the you are ready to begin work on the task. The experiment environment displays the Edit Menu.

Files

To complete a task, you may need to create, modify, or delete files. The experiment environment provides the following ways to manipulate files.

Create Files

The Create a New File option on the Edit Menu allows you to create a new file by specifying a filename for that file. For templates tasks, you are also prompted for the template you wish to instantiate. The new file created is a copy of that template file. If you wish to create a new file that is not a copy of a template (such as a test data file), choose the "blank" template from the list of templates.

Edit Existing Files

A list of existing files for this task is presented under the Edit an Existing File option, which allows you to select and edit any file listed.

ReinstantiatingTemplate Files

When modifying a previously-instantiated template, it is often convenient to start with an uninstantiated template, using "cut and paste" to instantiate this template based on the previously instantiated template. The Reinstantiate a Template option on the Edit Menu provides a mechanism for this. When you select this option, you are prompted for a new filename and a template to instantiate, just as with the Create a New File option. You are also asked which previously-instantiated template (from the inventory) you wish to view. You then enter emacs--the previously instantiated template is available in a read-only buffer, and the new uninstantiated template is available in a writable buffer.

Deleting Files

To delete a file from the list of files available for this task, choose the Delete an Existing File from Inventory option. Files that are not needed for compiling or testing a message should be deleted from the inventory before the task is closed.

RCS

If you wish to maintain versions of your files with RCS, use the RCS check-in and RCS check-out menu options. With both options, you are prompted for the file to check in or check out. With the RCS check-out option, you are also prompted for the revision you wish to check out.

Code and Test

You should now design, code and unit test. You should not modify the automatically-inserted comments at the top of the files. If you find during the course of modification that you no longer require one of the files for this ICD, please delete that file from the inventory.

Compiling MSL Programs

Rapid prototype compilation (to SML) of MSL programs is done with the Compile option from the Edit Menu. After a program has been compiled with this option, the rapid prototype testing options can be executed.

Ada code can be generated from an MSL program by choosing the Build option from the Edit Menu. After a build has successfully executed, the Ada Acceptance Test option on the Test Menu is available for execution. Since the build process is time consuming, this menu option prompts if you wish to start the build process overnight. If you choose this option, the build process will start after you exit the environment. After electing to run the build overnight, you can suspend the task and start a new task. However, when you return the next day please reactivate this task so it can be completed as soon as possible.

Testing MSL Programs

After an MSL program has been compiled with the Compile option, the following options are available on the Test Menu:

RP Interactive Test	Allows you to interactively enter and test messages and masks to test a program.
RP Unit Test	Executes all test cases in a test data file.

Compiling Templates

Instantiated templates can be compiled using the Compile option on the Edit Menu. This option compiles your program and places it in the library. The contents of the library can be examined with the Library Contents Listing option on the Edit Menu.

Before acceptance testing can be performed, the Build option must be executed. You are prompted for the instantiated template you wish to build. This template must be an instantiated version of one of the model solution test drivers (see *Model Solution Test Drivers*, below).

Testing Templates

Two menu options on the Test Menu allow you to test instantiated templates. The Compile-and-run option prompts you for the file you wish to compile and the name of the main program. After compiling this file and creating an executable, it will run the executable. This can be used for testing instantiated typecasters.

The Compile-and-run with data option can be used to test complete implementations of a message prior to running acceptance testing on the implementation. This option can only be run after the Build option has successfully executed. You are prompted for the filename of a test data file you wish to test. This file is in the same format as test data files for MSL tasks.

Model Solution Test Drivers

Drivers are provided for performing acceptance testing or testing with a file of test data. Each driver is a template that must be instantiated before it can be used for a particular task. One driver, `model_driver_ascii.a`, can be instantiated for testing character-based messages, and the other, `model_driver_binary.a`, can be instantiated for testing bit-based messages. Each driver needs to have three values defined--the name of the package defining the ICD typecaster, the name of the package defining the record typecaster for the message, and the maximum width of a message. The maximum width must be expressed in bytes, even if the message is bit-based.

Acceptance Testing

When you believe that your implementation is complete and thoroughly tested, you are ready for acceptance testing. The system will run a set of test cases for your message on your implementation and report the results. If there are discrepancies between the expected result and the actual result, those will be displayed for you.

Information will include all test results and an indication of where expected results differed from actual results. For each case, the expected action will be indicated, and the user representation (if appropriate) and the external representations of the message will be printed. Acceptance testing may be done as often as you desire, however, each time you attempt acceptance, you should believe that—to the best of your knowledge—the implementation is complete and correct.

If all acceptance tests are passed (ie: all good messages are accepted and translated, and all bad messages are rejected), then you are ready to complete the task.

If any acceptance tests failed, then you should identify the problem and perform defect repair on the implementation. Any work on the implementation after the first acceptance test has been run is considered "rework".

Acceptance Testing for MSL Tasks

The RP Acceptance Test option on the Test Menu executes the acceptance test suite on a program that has been compiled with the Compile option. When all test cases pass rapid prototype acceptance testing, the Build option can be executed to generate Ada code from the MSL program. After the Build option has been successfully executed, you can run the Ada Acceptance Test option on the Test Menu. All acceptance test cases must pass when run on the compiled Ada code before a task can be completed.

Acceptance Testing for Templates Tasks

Acceptance testing can be executed by choosing the Acceptance Test option on the Test Menu. This option will only work if Build has been successfully executed. All acceptance test cases must pass before a task can be closed.

Complete Task

When acceptance testing is clean, then you should select the Complete Task option on the Main Menu. The completion procedure verifies that acceptance testing has been successfully completed. All files will be checked-in to the system. The check-in procedure inserts a comment in each file containing the task identifier and a date/time stamp.

Task Check-In

After task completion is done, you should inform your manager who will note the completion date and time. You must then complete a Task Assessment Form for this task. Once again, careful thought and evaluation at this stage of the task are vital for valid experimental results. You are now free to begin the next task.

IV. Work Flow

The environment tracks the current status of all tasks. If you are in the middle of a task and need to stop work on the task for a period of time (i.e. for a lunch break or when you go home), you must exit the experiment environment by choosing the Quit option on the Main Menu. When you re-enter the experiment environment and want to continue working on the task, then you must choose the Continue with a Task option. The environment allows work on only one task at a time.

At some point it may be necessary to suspend work on a task, such as when a system bug is being fixed or the time limit for a task has been exceeded. This can be done by choosing the Suspend Current Task option on the Main Menu, which prompts you for the reason you need to suspend the task. After suspending a task, another task can be started. When you wish to restart a suspended task (and no other task is active), choose the Reactivate Previously Suspended Task option on the Main Menu.

V. Communication

It is vital to the conduct of the experiment that you do only your work, and do not gain much information about the work of other subjects. You may assist other subjects with technical questions, but do not inquire about details of the work. Do not, under any circumstances, share the details of a task ICD with another subject.

VI. Problems

You will probably encounter problems or ambiguities during the course of work. The experiment monitors are available to answer any questions, no matter how small. Please feel free to contact a monitor about any aspect of your job. The experiment monitors are:

Laura McKinney	mckinney@cse.ogi.edu	(503) 690-1450	10 AM - 3 PM EST
Jef Bell	bell@cse.ogi.edu	(503) 690-1482	3 PM - 8 PM EST
FAX		(503) 690-1548	

You will be notified in advance if this schedule changes.

Appendix B Participant Consent Form

Facilities:	Pacific Software Research Center and Intermetrics, Inc.
Program:	Air Materiel Command Contract No. F19628-93-C-0069
Experimenters:	Dr. Richard Kieburtz, Dr. James Hook

I consent to participate in this experiment designed to compare software methodologies. I have been informed that I will be asked to solve problems using two different development methods in the Message Translation and Validation domain, and that I will be required to develop new code and to perform maintenance work on existing code developed by other subjects. I understand that my work will be used in maintenance activities by other subjects.

I have been informed of the attached policies regarding communication with other experimental subjects.

I understand that my work will be monitored, and that the purpose of the monitoring is to determine the effectiveness of the two methodologies, and not to evaluate individual subject performance. I have been informed of reporting requirements which are designed to solicit subjective information about the use of the methods.

The results of this experiment will be reported in ways that preserve the anonymity of subjects. Information about subjects other than that recorded during the process of monitoring work and that reported by subjects on reporting instruments will be disclosed only to the investigators and their collaborators, and not to anyone uninvolved in this research project, specifically not to Intermetrics management.

I understand that the investigators and experiment monitors will be glad to answer any questions I have about this experiment. Questions which may influence the outcome of the experiment may be deferred until the end of the experimental period. Concerns about any aspect of this study may be referred to Dr. James Huntzicker, Chair, OGI Committee on Human Subjects. I understand that I may decline to answer any question or to engage in any procedure, and that I may withdraw from the experiment at any time for any reason.

I have read and understood the above, and I voluntarily consent to participate in this experiment.

Participant's signature Date

Name:
Address:

Phone:

I certify that I presented the above information to the participant.

Experimenter Date

Appendix C Proof-Of-Concept Experiment Manager's Weekly Report

Date: __________

Reporting Manager: ______________

Have there been any incidents impeding progress on subjects' tasks that are outside of the scope of the experiment (system downtime, company meetings etc.)?

Have all subjects been making progress on their individual tasks (identify any blockages or individual productivity drops due to illness etc.)?

Please identify any other problems or concerns you may have:

Appendix D Subject Supervisor's Worksheet

Check-out Chart

ICD Check-Out														
Indicate check-out of task by placing the subject ID in the corresponding box and the date and time in Out.														
Indicate completion of task by entering the date and time in In.														
In order to check out a task, all prior tasks must be complete.														
ICD	**Technology**	**Original**			**Mod 1**			**Mod 2**			**Mod 3**			**Mod 4**
		Subject	Out	In	Subject	Out	In	Subject	Out	In	Subject	Out	In	Subject
1	msl													
2	msl													
3	msl													
4	msl													
5	msl													
6	msl													
7	msl													
8	msl													
9	msl													
10	msl													
11	msl													
12	msl													

Appendix E SDRR Experiment: Monitor Process

Responsibilities

Monitors are primarily responsible for:

- subject motivation
- identification of problems
- resolution of problems
- communication with Intermetrics management
- manual report collection

Tracking

All communication between monitors and subjects after the start of the experiment should be tracked in order to ensure continuity and to provide a basis for analysis of problems. Monitors should maintain a written description of every conversation with subjects on the monitor log sheet. If problems are identified, the monitor should submit a problem report using Gnats with the category EXPERIMENT.

Subject Reporting - Timesheets

Subjects must fax a timesheet on Friday afternoon. Monitor is responsible for reviewing the timesheets for completion and relaying information.

Subject Reporting - Weekly Report

Subjects must also fax a weekly report. Monitor is responsible for collecting and reviewing the information. If problems are identified, monitor should submit a problem report. Monitors should telephone weekly with verbal feedback to maintain motivation. Monitors should also watch for schedule problems.

Problem Identification

-- system bugs
-- test suite challenges
-- inability to complete within time allotment
-- out-of-scope change
-- ICD ambiguity

Appendix F SDRR Experiment: Test Case Review Process

Goals:

The test case review process is designed to accomplish the following:

- Perform a consistency check on the ICD's and modifications generated by Hanscom
- Generate acceptance test cases for the task
- Assign a task identification number and package the ICD for the subjects

The process is designed as much as possible to avoid possible experimental bias by OGI personnel.

Timeline

A test case review will be held for each original ICD and modification approximately two weeks in advance of anticipated need. For the originals, this will occur two weeks prior to experiment start.

Personnel

A minimum of three people must review each item. Experiment monitors are explicitly exempted from participation due to the possibility of unintended communication with subjects regarding details of test cases.

ICD Review

A task identification number is assigned to the ICD by the team leader. A pair of team members will be identified to review each test case. One will be responsible for implementing the message in MSL and identifying a suite of test cases designed to exercise all aspects of the message - both positive and negative tests. The MSL program should be run through the test generator, and test messages collected. The messages should be tested with the program, and manual modifications made to handle interfield constraints. The buddy then needs to perform an inspection of the MSL program, the test suite, and the test data to ensure correctness and coverage of the message space. If there are any errors or discrepancies detected what is submitted by Hanscom, one member of the team will contact Harry Koch for instructions. Comments to eliminate ambiguities in the ICD will be added to pass along to the subjects.

Packaging

A member of the review team should then package the test data into the acceptance testing framework, and submit any modifications or elaborations on the ICD to Laura for inclusion in the packet for the subject.

Out-of-Scope Changes

If the ICD represents an out-of-scope change for MSL, then test suites will be identified and generated by hand. Out-of-scope changes must be handled by submitting an out-of-scope change request to the developers, describing the change to be made. The out-of-scope change process must then be followed by those implementing the change. The new code able to handle the out-of-scope change must not be submitted for use by the experiment subjects until they have explicitly identified the change and have requested an out-of-scope modification. Project monitors should not be involved in the details of out-of-scope changes and should not know which test cases involve such changes.

INTERFACE CONTROL DOCUMENT

Message Format Description

Message Title: Track 170

Modification:

Character Code: ASCII

Message Length: 50 or 52 characters

Function/Purpose: Message transmission.

Source: ROCC/SOCC

Classification: Unclassified

Version: 1.01

No.	Field Name	Size (chars)	Range of Values	Amplifying Data	
1	Message Identifier	3		3 character identifier	
	first character	[1]	A	Actual report	
			X	Exercise report	
	second character	[1]	T	Constant "T"	
	third character	[1]	H	Hostile (not legal if the first letter is X)	
			U	Unknown	
			K	Faker	
			S	Special	
			Y	Yoke	
			B	Bee	
			P	Pending	
			E	E-3	
			R	OTH-B uncorrelated	
			I	Interceptor	
	separator	1	/	Slash	
2	Track Designator	5	aa	Reporting and originating source designators followed by numerical designator	
	reporting	[1]	F	SW SOCC	
			R	NW SOCC	
			Z	AK ROCC	
			W	CW SOCC	
			S	CE SOCC	
			B	NE SOCC	
			C	SE SOCC	
			O	ICE ROCC	
			D	DEW	
	originating source	[1]	J	AEW&C in SW SOCC area	

No.	Field Name	Size (chars)	Range of Values	Amplifying Data	
			Q	AEW&C in NW SOCC area	
			M	AEW&C in AK SOCC area	
			H	AEW&C in CW SOCC area	
			X	AEW&C in CE SOCC area	
			K	AEW&C in NE SOCC area	
			N	AEW&C in SE SOCC area	
			I	AEW&C in GIUK area	
			L	PIN DEW sector	
			U	CAM DEW sector	
			E	FOX DEW sector	
			P	DYE DEW sector	
			T	GIUK	
			A	ECRS	
			V	CRS	
			Y	WCRS	
				If the first character is D, the second character must be one of the DEW designators	
	numerical designator	[3]	001-999	Assigned by originating source	
	separator	1	/	Slash	
3	Time	4		Time Group	
	hour	[2]	00-23		
	minute	[2]	00-59		
	end of line	1	CR	Carriage return	
4	Latitude and Longitude				
	latitude				
	degrees	[2]	00-89		

No.	Field Name	Size (chars)	Range of Values	Amplifying Data	
	minutes	[2]	00-59		
	direction	[1]	N or S	North or south	
			blank	Not reported	
	separator	1	/	Slash	
	longitude				
	degrees	[2]	000-179		
	minutes	[2]	00-59		
	direction	[1]	E or W	East or west	
			blank	Not reported	
	separator	1	/	Slash	
5	Other Actions	2	aa	Two character descriptor	
	first character	[1]	A	Active. Radar data supports track	
			N	No data. No radar data supports track	
			S	Simulated. Simulated track	
	second character	[1]	E	ECM track	
			F	Friendly	
			M	Maneuvering	
			H	High mach	
			X	Maneuvering & High mach	
			R	Request	
			S	Spawned	
			D	Military decision	
			K	Splash	
			Z	Spawning	
			P	Passes to adjacent ROCC	
			Q	Track has exited	
			C	Clear	

No.	Field Name	Size (chars)	Range of Values	Amplifying Data	
	end of line	1	CR	Carriage return	
6	Course	3	001-360	In degrees	
			"000"	No value reported	
	separator	1	/	Slash	
7	Speed	4	0000-5110	In knots	
	separator	1	/	Slash	
8	Altitude or Track Confidence	0 or 2	01-99	In thousands of feet	
			HH	High confidence	
			MM	Medium confidence	
			LL	Low confidence	
			NN	No confidence	
			blank	No altitude value reported or altitude less than 1000 feet	
	separator	1	/	Slash	
9	Number of Objects	2	00-99	Actual or estimated number of airborne objects associated with this track	
	end of line	1	CR	Carriage return	
10	Weapons Committed	2	01-99	Number of weapons committed	
			AD	Action deferred	
			NF	No fighters	
			CR	Out of range	
			WX	Weather	
			00'	No value to report	

No.	Field Name	Size (chars)	Range of Values	Amplifying Data	
	separator	1	/	Slash	
11	Accumulated Kills	2	01-99	Number of kills reported on this track	
			00'	No value to report	
	end of line	1	CR	Carriage return	

Appendix H   Task Effort Allocation

Allocating your hours of effort for each particular task is important for valid experimental analysis. Please be aware that your time allocations are not reported to your management and are kept strictly confidential. It is vital that you be completely honest in reporting actual effort hours expended, whether or not you end up with 6 or 10 hours a day is not of interest to the experiment monitors.

Please keep the task effort allocation sheet out on your desk in an accessible location, and record effort spent **at the time you are working**. Reconstructing effort spent at the end of the week is error-prone and will compromise the validity of the results.

Time on a task is considered either "development" time or "rework" time. Development time is all time spent up until acceptance testing is run on that task for the first time. Then all subsequent time is considered "rework" time. (Note that a modification of an ICD will have both development time and rework time.)

Below is a list of activities that **should** be recorded as effort on a particular task:

- analysis to generating estimate of hours to completion
- code and unit test
- defect repair
- waiting for the system to run that could not otherwise be spent in productive activity on another task
- receiving help from experiment monitors in understanding the ICD
- reviewing technical material or reading reference manuals
- asking technical questions about either technology
- browsing template files
- understanding existing implementations

Below is a list of activities that should **not** be recorded as effort on a particular task:

- administrative tasks such as check-in, check-out, and task evaluation
- completing weekly reports and task allocation sheets
- communicating with monitors regarding problems in the experiment environment
- waiting for defects in the system to be repaired

If you are doing something that you do not see listed above, please ask a monitor for clarification about how the time should be allocated. If you have any other questions about how your time should be recorded, please contact a monitor for assistance.

Task Effort Allocation

Subject ID: ________

Week: ________

Please record hours daily and fax the sheet to the experiment monitor on Friday afternoon. Timesheets are confidential. Please record only actual time spent working on the task.

Date	Task Identifier	Hours	Status*

*Status is either "development" or "rework", where "rework" is defined as any effort after the first acceptance test is run.

Appendix I Proof-Of-Concept Experiment Task Assessment Form

Subject ID: ____________

Date: ____________

ICD: ____________

Technology: ____________

Part I

Implementation Difficulty
Please place a vertical bar at the point on the line which represents the difficulty of implementation of this task.

|__________|__________|__________|__________|

0 25 50 75 100

Easiest Most Difficult

Implementation Completeness
Please place a vertical bar at the point on the line which represents the percent of completeness of your implementation of this task.

|__________|__________|__________|__________|

0% 25% 50% 75% 100%

Totally Incomplete Complete

Part II

Please *circle the appropriate number* for each statement, based on the scale below.

1	2	3	4	5
Strongly Disagree	Moderately Disagree	Neither Agree nor Disagree	Moderately Agree	Strongly Agree

Control of Work

While working on this task, it was easy to determine which parts of a task were complete.

1 2 3 4 5

While working on this task, it was easy to determine which parts of a task remained to be completed.

1 2 3 4 5

While working on this task, it was difficult to move from one subtask to another.

1 2 3 4 5

Interface

The user interface was convenient to use.

1 2 3 4 5

The uesr interface limited me in the way I did work on this task.

1 2 3 4 5

The user interface helped me to avoid making errors in this task.

1 2 3 4 5

Flexibility

It was difficult to make changes to this implementation.

1 2 3 4 5

It was easy to locate errors.

1 2 3 4 5

It was easy to repair errors.

1 2 3 4 5

Performance

The system performed to my satisfaction during my work on this task.

1 2 3 4 5

Part III

Please answer the following questions:

What were the nature of performance problems you encountered, if any?

Did you encounter situations where data or work was lost or corrupted?

Do you have any specific comments on the usability of the system during the course of completing this task?

Do you have any other comments or questions about the system?

Pacific Software Research Center
January, 1995
Version 1

Debriefing Phase I
Conduct of the Experiment

During this phase of the debriefing, we would like you to focus on the logistics of the experiment rather than aspects of either technology. You will have an opportunity to talk in detail about the different methods in the next phase of the session. The goal of this phase is to identify aspects of the conduct of the experiment that may affect the analysis of the data. If there are any issues that are not raised by the questions, please feel free to comment. Your privacy will be protected as all answers will be identified solely with your subject identifier. Please be honest -- no performance evaluation will be made based on your answers in this debriefing.

I. Communication

A. We anticipated that some degree of communication amongst subjects would normally occur over the duration of the experiment. Please characterize the content, duration, and participants in typical interactions?

B. As we indicated in the beginning, we expected subjects to require technical assistance over the course of the experiment. Did you take advantage of technical assistance from other subjects, your management, the OGI experiment monitors, the model solution technical trainer, or any other personnel during the course of the experiment? If so, please characterize the type and extent of the technical assistance.

C. Did you at any time require technical assistance that was not available? If so, please explain.

D. Were the experiment monitors available and responsive during the experiment? If not, please elaborate.

E. Do you have any other thoughts on communication during the course of the experiment?

II. Experiment Environment

Questions below may refer to the experiment environment. This is the menu-driven interface you were asked to use during the course of the experiment. Please focus on the interface and not aspects of the technologies themselves, as they will be discussed separately.

A. Did the experiment environment impede work in either method? If so, please elaborate and indicate how much of an effect the environment had on your personal productivity.

B. Did the experiment environment testing process give you sufficient support and feedback during the testing process?

C. Did you encounter defects when using the experiment environment? If so, please describe and indicate how much this affected your productivity.

D. What percentage of your time did you spend off-line (not logged in under the experiment environment) while still working on tasks?

E. Do you have any other comments on the experiment environment?

III. Process

Questions below refer to the experiment process. These encompass the procedures you were asked to follow. Please focus on the processes themselves.

A. Did you encounter problems with incorrect acceptance test data? If so, how much time did you spend discovering the error(s)?

B. Did you have an opportunity to assess the coverage of the acceptance test data? If so, what is your opinion of the coverage of this automatically-generated data?

C. Characterize the type and amount of time you typically spent in overhead tasks not directly related to a particular technology. (Such tasks might include running acceptance tests (but not debugging), compiling, reading and understanding the ICD.) Please indicate if there is a difference in time spent between the two technologies in amount of overhead required.

D. Were your tasks sufficiently defined (ie: was there enough information to evaluate whether you were completing tasks correctly and sufficiently) with regard to commenting, data structure definition, and criteria for display of the user representation of the ICD?

E. Did you have problems with ambiguities or difficulties in understanding ICD's? If so, please describe the types of problems you encountered and at what stage in the development process they were discovered and resolved.

F. Do you have any further thoughts on the processes you were asked to follow during the experiment?

IV. Training

A. Was the training you received adequate for the tasks you were asked to perform? If not, please indicate what training you needed but lacked. Please consider all training, including technology and experiment environment training.

B. Considering each technology individually, how long did it take until you felt fully competent? Did you reach this point before or after the start of the experiment?

C. Please indicate your level of satisfaction with the training you received on both technologies and the experiment environment itself (very satisfied, satisfied, unsatisfied, very unsatisfied) and why.

D. For each trainer, please describe how you felt about the trainer's teaching (very positive, positive, satisfied, negative, very negative) and why.

E. Do you have any additional comments on training?

V. Reporting

A. How would you characterize the accuracy of your reporting of time spent on each task? (Please express your answer in a unit of time - eg: to the half-hour...)

B. When did you complete time sheets (after each task, daily, or weekly)?

C. Do you have any other thoughts on the reporting you were asked to do? In particular, were there questions or items you would have wanted to report but were not provided the opportunity to do so?

VI. Personal

A. Did you experience boredom or feel that the tasks were not challenging or stimulating at any time during the experiment? If so, please explain.

B. What were your primary motivations during work on tasks? What "kept you going"?

C. Please describe your most positive experience during the experiment.

D. Please describe your most difficult or negative experience during the experiment.

E. Do you have any additional comments about any aspect of the conduct of the experiment?

Debriefing Phase II
Comparison of the Two Technologies

During this phase of the debriefing we would like you to focus on the two technologies. If comments on the conduct of the experiment occur to you, please make a note and return to the noted items at the end of the debriefing, where you will be allowed time to comment on any additional thoughts you may have had.

1. Predictability

a. How much confidence did you have in your ability to accurately predict the difficulty of implemention? Please characterize your confidence in prediction for each of the following criteria:

Original ICD using Model Solution

Original ICD using MSL

Modification of ICD using Model Solution

Modification of ICD using MSL

b. What contributed to the ease or difficulty of prediction for each of the above.

c. Was it easy to determine if a particular ICD contained specifications of items whose implementation was beyond the limitations of the existing Model Solution technology (eg: required writing new template or modifying the generics)? Please explain why or why not.

d. Was it easy to determine if a particular ICD contained specifications of items whose implementation was beyond the limitations of the existing MSL language (eg: could not implement without changes to the language)? Please explain why or why not.

e. Please answer with respect to each technology individually: Did your ability to accurately predict time to completion improve over the course of the experiment? If so, how long before you became proficient?

f. Please respond with respect to each technology individually: What factors made it easy or difficult to predict time-to-completion?

2. Usability

a. In which method did you have most confidence in the correctness of your solution? Why?

b. During the maintenance cycle (ICD modifications), which technology provided implementations that were easy to make maintenance changes? Please explain.

c. Please respond for each technology individually: What made this method easy or difficult to use?

d. Which method would be easier for you to train someone else to use? Why?

e. If you were to return to this project 6 months later to make maintenance changes to implementations, which implementation would you choose to work with: the MSL or the Model Solution? Please elaborate.

f. If you were required to use an implementation of an ICD from this experiment in a real system with expected maintenance, would you choose to use the Model Solution implementation or the MSL implementation? Why?

g. Please respond for each technology individually: If the ICD were lost and you were required to reconstruct the ICD from the implementation, how difficult would this be? Why?

h. In which technology is it easier to document a solution? Why?

i. In which technology will it be easier to maintain your skill level without retraining? Why?

j. Please answer for each technology individually: How easy or difficult was it to trace the source of an error?

k. Please answer for each technology individually: How easy or difficult was it to correct errors once located?

l. Programming is a human process with much opportunity to introduce errors. Which technology had the least opportunity for allowing error introduction? Why?

m. For each technology individually, please characterize the typical type of error made or problem discovered.

3. Adaptability

a. Which technology better accommodated the entire range of ICD's used in the experiment? Why?

b. What are the limitations of each technology in its application to ICD's?

4. Flexibility

a. Please answer for each technology individually: Did you find it easy to locate out-of-scope changes (those for which the existing technology was inadequate)?

b. For each technology, what were the out-of-scope changes you encountered? Please assess the difficulty of implementing out-of-scope changes.

c. Which technology allowed you to write programs that were most faithful to the ICD?

d. For each technology, which requirements were you unable to implement completely?

5. Productivity

a. In which technology do you feel you were most productive using in implementing original ICD's? Why?

b. In which technology do you feel you were most productive using in implementing modifications to ICD's? Why?

c. Which technology took longer to fully learn? Please explain.

d. Please respond for each technology individually: Where could you get further productivity gains?

e. Please answer for each technology: In what activity was the majority of your development time spent?

f. Please respond for each technology individually: Do you think a graphical interface would have improved your productivity?

g. For MSL, did you have problems correcting type errors (e.g.: using a reader of the wrong type)?

6. General

a. What would your ideal solution to the Message Translation and Validation domain look like?

b. Please respond for each technology: What level of formal education and training would be needed for an individual to be able to use the technology productively?

c. Do you have any other comments regarding the differences or aspects of the technologies?

Accel Model Solution MTV Questions

7. Training Effectiveness

a. How well did the course, *The MTV™ Model, A course for application developers*, prepare you for the tasks you had to accomplish during the experiment?

b. Looking back, what areas should have been concentrated on more? What areas should have been concentrated on less? What areas could be improved and how?

c. How helpful was the presentation on the first day that provided the foundation and context for software models in general and the MTV model more specifically? (We described traditional engineering and its application to software engineering in the form of Model-Based Software Engineering (MBSE).)

8. Model Solution MTV Technical Maturity

a. How appropriate were the concepts provided by the MTV model (typecasters, external representation translator and validator, message representations, etc.) for forming the message handling problems you encountered in the experiment? That is, did the concepts help you see how the problem could be set and solved using the MTV model without actually doing it?

b. How effective were the specification forms provided with the MTV model for setting the message handling problems you encountered in the experiment? What percentage of the messages/fields encountered were you able to specify using the forms? How could the forms be improved and/or extended to handle the messages that you could not specify? Do you have any suggestions for better message specification forms?

c. How effective were the code templates provided with the MTV model for solving the message handling problems you encountered in the experiment? What percentage of the message/fields encountered were you able to solve using the templates? How could the code templates be improved and/or extended to handle the message that you could not solve? Did you create or see the need to create any new Ada generic typecasters and code templates to handle new classes of fields?

d. Did you experience any problems with or bugs in the MTV model code? Please provide details.

e. Do you have any suggestions for improving the model concepts, forms, templates, and/or software architecture/structure? (i.e., resulting Ada *with*ing dependencies)?

f. After generating the MTV Ada code, how easy or difficult do you believe it would be to integrate this module with an existing system?

9. Accel MTV Product Commercialization

a. Do you think the Accel Model-Based MTV technology has commercial potential?

b. Accel believes a graphical user interface providing message specification, message management, and code management capabilities is essential to moving this technology to the commercial market place. Accel is in the early stages of developing a tool (*MTV Builder™*) for automating specification of message formats and generation of Ada code based on the MTV model.

Based on you experiences manually applying the MTV model, do you have any insight or suggestions for the *MTV Builder* tool (development capabilities, interface, plug-and-play capabilities, etc.)? What would such a tool look like to you (brainstorm, dream, the sky's the limit!)? What kind of tool would make your life as a model-based application developer fun by allowing you to address the true challenges of system development and easy by automating the mundane parts of application development?

c. What are the markets for the *MTV Builder* tool? How would you position the product? What three things would you stress in marketing this technology/tool?

SDRR MTV-G Questions

The following questions are specific to the use of the Software Design for Reliability and Reuse (SDRR) method. The SDRR method utilizes domain-specific languages coupled with program generators to produce software modules from specifications in a domain language.

This experiment applied the SDRR method to a particular domain - the Message Translation and Validation (MTV) domain. The domain-specific language developed was Message Specification Language (MSL), and the program generator developed produced executable Ada code for message specifications expressed in MSL. The program generator is referred to as MTV-G, for Message Translation and Validation Generator.

a. Do you prefer the use of the domain-specific languages (such as MSL) and the use of a generator (such as the MTV-G) to produce implementation language code over traditional software development methodologies you have used in the past ?

b. What is the most difficult part of using a domain-specific language and a software generator?

c. What are the advantages and disadvantages of using a domain-specific language and a software generator over traditional software development and maintenance methods ?

d. How difficult would it be to institutionalize the use of SDRR-based generators within your company ? How difficult would it be to institutionalize the development and maintenance of SDRR generators within your company?

e. From a software maintenance perspective, how well does maintenance on implementations developed under SDRR (such as MSL specifications) measure up to other maintenance processes you have used in the past ? i.e. much worse, worse, about the same, better, much better?

Phase III
Wrap-Up

A. Do you have any other comments on either the technologies or the conduct of the experiment?

Thank you very much for your participation. OGI will make copies of the analyses and papers written based on this experiment available to all of the subjects. If you would like further details on the on-line monitoring environment that was used and how your work was tracked, please feel free to call either Jef Bell or Laura McKinney at any time. If you have other concerns such as privacy, please contact us.

At this time the debriefing managers will leave the room. You may answer the questions on the attached sheet with your subject ID, seal in the envelope provided, and mail to OGI.

Subject Identifier: A

Date of Debriefing: ____________

Debriefing
Your responses to these questions will be kept confidential.

A. Did you feel comfortable in answering the debriefing questions? Why or why not?

B. Were there any comments about your management and the conduct of the experiment that you would like to make? Please explain.

Pacific Software Research Center
January, 1995
Version 1

Group Debriefing

Please ensure that all participants have a chance to respond to each question. Allow for interchange and discussion on each question. Allow participants to return to previous questions if they have further thoughts.

1. Please describe your understanding of how the SDRR Method (the use of domain-specific specification languages coupled with program generators) for software development works.

2. Please describe your understanding of how the Model-Based Solutions approach to software development works.

3. What do you see as the potential benefits and limitations of the SDRR Method (the use of domain-specific specification languages coupled with program generators)?

4. What do you see as the potential benefits and limitations of the Model-Based Solution approach?

5. Which technology do you believe is most commercially viable and why?

6. Software engineering experiments attempt to model real-world environments while still providing a mechanism to measure and examine aspects of the software engineering process. There are trade-offs to be made between constraining the environment to the point where measurements can be made and making the environment more realistic. Each experimental design will have strengths and weaknesses in its attempts to model a genuine process.

a. From your perspective, what aspects of the software development process was this experiment designed to measure and evaluate?

b. What do you believe are the strengths of the experimental design -- what aspects can be generalized to real-world environments?

c. What do you believe were the limitations of this experiment in generalizing results to real-world environments?

Volume VII
Baseline Performance Measurements of Un-optimized Generated Code

Baseline Performance Measurements of Un-optimized Generated Code

Dino P. Oliva
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
PO Box 91000
Portland, Oregon 97291-1000, USA
http://www.cse.ogi.edu/PacSoft
Email: oliva@cse.ogi.edu

February 27, 1995

1 Description

The Pacific Software Research Center (PacSoft) is currently finishing the first phase of a project developing the Software Design for Reliability and Reuse (SDRR) Method [B$^+$94, Kie95a]. The method is based upon generating software components from formal specifications. During the first phase of the project, an experiment was conducted to compare the SDRR method with a fielded technology. In this document, we compare the performance of the code produced in the experiment by both technologies.

2 The Experiment

As part of the proof-of-concept of the SDRR method, we compared our methodology in a message translation and validation (MTV) domain. This problem domain was chosen because there was a pre-existing, state-of-the-art solution to this problem based on code templates. The templates solution had already been proven to increase programmer productivity and would be a good test for our approach.

The SDRR Validation Experiment was then designed to compare both technologies during a typical software maintenance cycle (this experiment is described in detail elsewhere [Kie95b]). Both technologies were applied to a common set of tasks and revisions. The tables in this document compare the size and performance of code generated by the MTV-G with the code produced by the templates solution for the same task and revision.

In some cases the compiled code was not available for either the MTV-G code or the templates code. In these cases, the code size and execution time are reported as 0 and the `MTV-G/Templates` ratio is reported as `N/A`. Note that tasks 9-12 in the experiment had no revisions and hence no revision data appears in their tables. Limitations of MTV-G at the time of the experiment prevented the generation of Ada code for tasks 2, 3, and 8, so performance tables do not appear for them.

3 Performance Measurements

The size measurements reflect the size of the compiled executables. Size is given in bytes and was collected automatically using the standard unix command `ls -l`.

For the time measurements, we ran each of the executables on 500 test messages (250 positive, 250 negative). These messages were obtained by replicating the experiment test data. MTV-G code and templates code implementing the same task/revision were given the same input. Time is given in seconds and was collected automatically using the standard unix facility `time`.

3.1 Environment

All MTV-G code was generated using the Experiment baseline of MTV-G. The release numbers for each of the tools in the baseline are:

- ADL Translator 2.35
- HOT 1.16
- Lambda Lifter 1.21
- MSL Compiler 1.64
- Chin 1.23
- ML2Ada 1.75

All Ada code was compiled using SunAda version 1.1(j) at optimization level 4 (default). All of the code was tested on a SPARCstation 10 running SunOS 4.1.1.

4 Analysis

Averaging the ratios of the initial revisions of each of the tasks, the generated code averages about 1.7 times larger and 28.6 times slower than the equivalent templates code.

These ratios are quite encouraging. The goal of the first phase of the project was to develop a robust system that generated correct code and little consideration was given to the performance of the generated code. As such, many standard optimizations were left out that typically significantly improve performance. In the second phase of the project we are focusing on performance and these measurements provide an important baseline to track improvements of the generated code.

References

[B$^+$94] Jeffrey Bell et al. Software Design for Reliability and Reuse: A proof-of-concept demonstration. In *TRI-Ada '94 Proceedings*, pages 396–404. ACM, November 1994.

[Kie95a] Richard B. Kieburtz. Software Design for Reliability and Reuse—Method Definition Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1993.

[Kie95b] Richard B. Kieburtz. Results of the SDRR Validation Experiment. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1995. In [Pac95].

[Pac95] Pacific Software Research Center. SDRR Project Phase I Final Scientific and Technical Report. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1995.

Task	MTV-G		Templates		MTV-G/Templates	
Revision	Size	Time	Size	Time	Size	Time
0	1171456	121.4	516096	3.6	2.26984	33.7222
1	1064960	122.3	524288	3.7	2.03125	33.0541
2	1081344	117.8	524288	3.8	2.0625	31
3	1105920	124.8	532480	4	2.07692	31.2
4	1204224	135.7	0	0	N/A	N/A
5	1015808	116.4	0	0	N/A	N/A
6	0	0	0	0	N/A	N/A
7	0	0	0	0	N/A	N/A

Table 1: Task 1 Performance Measurements

Task	MTV-G		Templates		MTV-G/Templates	
Revision	Size	Time	Size	Time	Size	Time
0	720896	43.4	491520	3.6	1.46667	12.0556
1	688128	68.1	491520	3.5	1.4	19.4571
2	671744	62.9	475136	3.5	1.41379	17.9714
3	671744	64.6	475136	3.3	1.41379	19.5758
4	671744	65.8	483328	3.4	1.38983	19.3529
5	671744	62.3	483328	3.4	1.38983	18.3235
6	729088	61.9	491520	3.4	1.48333	18.2059
7	0	0	0	0	N/A	N/A

Table 2: Task 4 Performance Measurements

Task	MTV-G		Templates		MTV-G/Templates	
Revision	Size	Time	Size	Time	Size	Time
0	917504	55.9	516096	2.3	1.77778	24.3043
1	925696	57.5	491520	2.3	1.88333	25
2	933888	57.2	491520	2.4	1.9	23.8333
3	974848	83.9	507904	2.9	1.91935	28.931
4	991232	102.6	630784	9.9	1.57143	10.3636
5	1024000	111.4	614400	9.6	1.66667	11.6042
6	1228800	101.3	532480	3.5	2.30769	28.9429
7	1228800	99.5	532480	3.5	2.30769	28.4286

Table 3: Task 5 Performance Measurements

Task Revision	MTV-G		Templates		MTV-G/Templates	
	Size	Time	Size	Time	Size	Time
0	696320	40.3	540672	3.7	1.28788	10.8919
1	614400	36.7	540672	3.8	1.13636	9.65789
2	614400	37	540672	3.7	1.13636	10
3	622592	39.9	540672	3.8	1.15152	10.5
4	647168	41.6	548864	3.6	1.1791	11.5556
5	655360	41.2	548864	3.6	1.19403	11.4444
6	655360	41.8	548864	3.8	1.19403	11
7	0	0	0	0	N/A	N/A

Table 4: Task 6 Performance Measurements

Task Revision	MTV-G		Templates		MTV-G/Templates	
	Size	Time	Size	Time	Size	Time
0	1155072	109.7	557056	3.2	2.07353	34.2813
1	1056768	110.1	557056	3.3	1.89706	33.3636
2	1114112	113.1	565248	3.3	1.97101	34.2727
3	1114112	114.1	565248	3.3	1.97101	34.5758
4	1277952	167.2	589824	3.8	2.16667	44
5	1302528	168	598016	3.9	2.17808	43.0769
6	1318912	175.2	606208	3.9	2.17568	44.9231
7	0	0	0	0	N/A	N/A

Table 5: Task 7 Performance Measurements

Task Revision	MTV-G		Templates		MTV-G/Templates	
	Size	Time	Size	Time	Size	Time
0	737280	74.1	466944	2	1.57895	37.05

Table 6: Task 9 Performance Measurements

Task Revision	MTV-G		Templates		MTV-G/Templates	
	Size	Time	Size	Time	Size	Time
0	696320	201.8	458752	5.7	1.51786	35.4035

Table 7: Task 10 Performance Measurements

Task	MTV-G		Templates		MTV-G/Templates	
Revision	Size	Time	Size	Time	Size	Time
0	737280	81	401408	2.6	1.83673	31.1538

Table 8: Task 11 Performance Measurements

Task	MTV-G		Templates		MTV-G/Templates	
Revision	Size	Time	Size	Time	Size	Time
0	729088	76.9	425984	2	1.71154	38.45

Table 9: Task 12 Performance Measurements

Volume VIII
Technology Transition Plan

Table of Contents

Paragraph Page

Technology Transition Plan
Revision 8
28 February 1995

1. INTRODUCTION

The Pacific Software Research Center (PacSoft) of the Oregon Graduate Institute (OGI) of Science and Technology has developed a model for technology transition that will be applied to the Software Development for Reliability and Reuse (SDRR) tools and method (see Figure 1). This model has been designed to reduce the time and effort required to move the SDRR technology from proof of concept to common use. The basic premise for the model shown in Figure 1 is that technology transition activities and development activities occur in parallel. It is assumed that technology development occurs iteratively and passes throughout several overlapping and recursive phases, for instance:

a. Develop the basic technology that will implement SDRR.

b. Build a single instantiation of an application using the technology; in this case, the Message Translation and Validation – Generator (MTV-G).

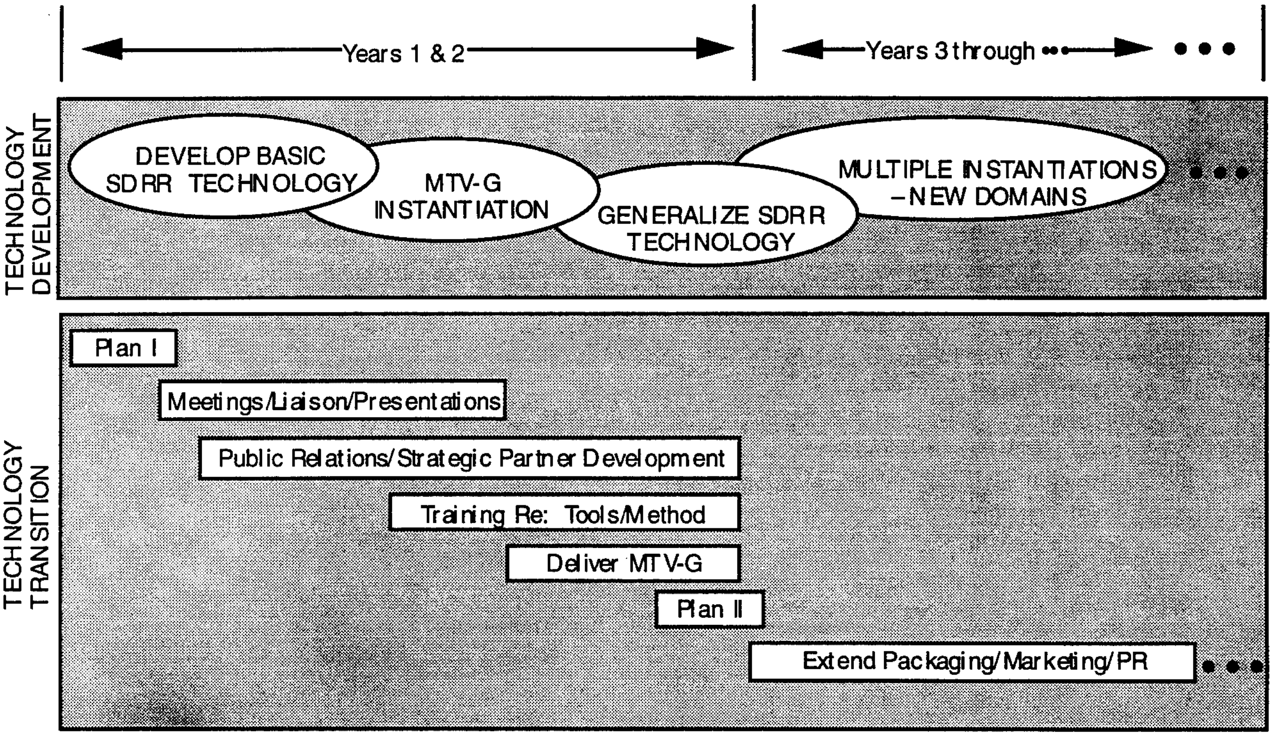

Years 1 & 2
Years 3 through •••
• • •
TECHNOLOGY DEVELOPMENT
DEVELOP BASIC SDRR TECHNOLOGY
MTV-G INSTANTIATION
GENERALIZE SDRR TECHNOLOGY
MULTIPLE INSTANTIATIONS –NEW DOMAINS
TECHNOLOGY TRANSITION
Plan I
Meetings/Liaison/Presentations
Public Relations/Strategic Partner Development
Training Re: Tools/Method
Deliver MTV-G
Plan II
Extend Packaging/Marketing/PR

Figure 1. SDRR Rapid Technology Transition Process

c. Generalize the technology to be applicable in more than one domain.

d. Create multiple instantiations of the SDRR method that can be applied to different application domains.

The first three sets of activities are planned to begin during the first two years of SDRR development; however, the third activity (generalizing the technology for multiple uses) will continue beyond year two into the long-term plans for the program. The fourth activity (creating multiple instantiations) will continue through the life of the SDRR technology.

In parallel with development, several technology transition activities are occurring. These activities are:

a. Planning the first phase of transition activities.

b. Attending professional meetings, developing professional liaisons with organizations that will act as agents for building visibility for the technology, and making presentations at professional meetings on the state and value of the technology.

c. Building a public image for the technology and developing strategic partnerships with organizations that will use the technology.

d. Preparing and delivering training in the SDRR tools and method.

e. Delivering and validating the MTV-G instantiation of the SDRR method.

f. Planning the second and subsequent phases of transition activities.

g. Implementing the follow-on technology transition plan.

This technology transition plan is presented in two parts to coincide with the overall program Master Plan and Schedule [WBS Task 5]. Part I details the technology transition activities during the initial technology maturation period (Years 1-2 shown in Figure 1). Since most of the effort for this period is concentrated on technology maturation, technology transition activities are limited to essentially preparing and packaging the SDRR products and method for it's initial transition into the Air Force's Portable, Reusable, Integrated Software Modules (PRISM) Program, training a core group of people in the use of the tools and method, publication and presentation of ongoing research results, and identifying other potential domains of interest and users.

Part II of this plan [WBS 5.7] will be prepared at the end of the technology maturation period according to the Master Plan and Schedule and will be delivered as a revision to this document. Part II will describe the long-term aspects (beyond the scope of the initial technology maturation period, Years 3 and beyond) of technology transition that will prepare the SDRR products and method to make the transition from development into common use.

2. RELATIONSHIP TO OTHER PLANS

This plan describes the technology transition activities that will be implemented as part of the SDRR Technology Development and Validation Program. Figure 2 shows the relationship among the plans for this project. The project overview and management approach is presented in the Management Plan. The Management Plan provides a description of the development process and Work Breakdown Structure to be used for the entire project. It also sets the context for the three parallel support plans (the Measurement Plan, Technical Plan, and Technology Transition Plan) that provide more detailed definitions of the work to be performed in these distinct program areas. Where other activities provide inputs to technology transition, the plans for those activities are referenced for the details of those activities.

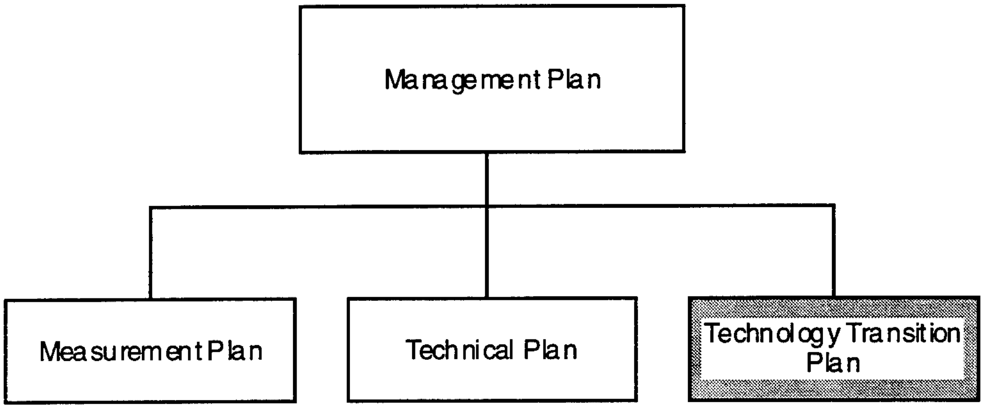


Figure 2. SDRR Project Plans

3. PART I - TECHNOLOGY MATURATION PERIOD PLAN - DETAILS

3.1 Responsibilities and Authority

The Pacific Software Research Center (PacSoft) of the Oregon Graduate Institute of Science and Technology assumes primary responsibility of the technology transition activities identified in Part 1 of this plan. AF/ESC assumes secondary responsibility for those technology transition activities where explicit AF/ESC responsibility is listed.

3.2 Work Breakdown Structure, Milestones, and Schedule

The WBS, Milestones, and Schedule for these activities is contained in the overall program Master Plan and Schedule [WBS Task 5]. The following paragraphs describe the major technology transition activities and milestones/events.

3.2.1 Technology Transition Planning

A technology transition plan, delivered in two parts, will describe the activities proposed to ensure that the SDRR technology makes a rapid transition from the university research environment into practice.

Part I will be prepared and delivered at the start of the technology maturation period and will describe the technology transition activities that will occur during the technology maturation phase of the effort.

Part II will be completed at the end of the technology maturation period and will describe the long-term aspects (beyond the scope of the initial technology maturation period) of preparing the SDRR products and method to make the transition from development into common use.

The PacSoft Center Director is responsible for this activity.

3.2.2 Meetings, Liaisons, and Presentations

Professional meetings and liaisons with existing technology transition agents[1].about the SDRR technology will be initiated. These interactions are expected to be informal information exchange forums. Attendance at the following meetings is planned:

a. The Software Engineering Institute's (SEI's) annual Software Engineering Symposium (August 1993) and subscribe to SEI membership.

b. The annual Software Technology Conference (STC) sponsored by the Software Technology Support Center (STSC) (April 1994).

[1] Martin, Cecil E., "Concept Paper on Structured Technology Transition, SEI, 1991.

Liaisons will be made with technology transition agents through attending meetings and making the following contacts:

a. Attend Advanced Research Project Agency (ARPA) contractors meetings in the area of software engineering (AF/ESC is required to initiate invitations). One meeting is tentatively scheduled in December 1993, others may be scheduled at a later date.

b. Meet with members of the PRISM Project at AF/ESC, Hanscom AFB (HAFB) to discuss plans for the technology validation experiment (more than one meeting may be required).

A technical presentation will be prepared for the STC in April 1994 to introduce SDRR to this community and highlight any research breakthroughs during the first year of the technology maturation phase.

In addition to the events listed above, more informal dialogues (E-mail, electronic bulletin boards (e.g., STSC, HAFB, Internet, MILNet, Athena) etc.), will occur as a natural by-product of the research with other researchers, technology experts, domain experts, the European ESPRIT community, and domestic software engineering interest groups).

All technical staff will participate in these activities.

3.2.3 Public Relations, Strategic Partner development

The public relations activities are marketing oriented. They generally consist of the effort required to prepare and distribute media and demonstration materials for the purpose of building advocacy for the tools and methods, and for building strategic partnerships. During this effort:

a. An informal market survey will be performed in order to identify potential users, domains of interest, and strategic partners for the use and commercialization (production of commercial-grade tools) of SDRR. The results of this survey will be included in the Final Technical Report. It is also expected that these results will also impact the Part II Technology Transition Plan.

b. PacSoft will initiate an affiliate membership drive to encourage commercial affiliations with the hope of establishing strategic partnerships as early as possible that will provide both resources and support to the ultimate commercialization of this technology. PacSoft will initially target local software houses and software tool developers (e.g., Cadre, Mentor Graphics, HP).

The Center Director is responsible for this activity.

3.2.4 Training in SDRR Technology Tools and Methods

This activity provides formal training materials and training. The task is essential to, and prerequisite for, the technology validation experiment. For this experiment (see the Technical Plan), independent contractors will be trained to use the tools and methods. There are two parts to this task.

a. Formal training materials will be prepared to supplement project technical documentation as an introduction to the tools and methods required to generate and maintain software using SDRR.

b. The two independent contractors will be trained in the use of the tools and methods required to generate and maintain software using SDRR.

An expected side effect of technology development is that several PacSoft staff members and students will become trained in the SDRR method as a natural by-product of the research. This is a key factor in building the base of SDRR knowledgeable people.

The Project Manger will be responsible for this activity.

3.2.5 Deliver and Install MTV-G and SDRR tools in the PRISM Program Environment

The independent contractors trained by PacSoft (Paragraph 3.2.4) will deliver and install the MTV-G software and some subset of the SDRR tool set (as needed) in the PRISM Program environment for the conduct of the experiment defined in the Technical Plan. This effort will be coordinated with AF/ESC.

The Project Manger will be responsible for this activity.

3.2.6 Prepare and Document Lessons Learned

Technology transition lessons learned will be compiled and documented for inclusion in the Final Technical Report

The Project Manger will be responsible for this activity.

3.3 Technical Reviews

A report on the progress of the technology transition activities will be included at each scheduled PMR. Measures of technology transition success will be reported in accordance with the Measurement Plan.

3.4 Program Risk Analysis

Technology transition risks and mitigation plans are included in the Management Plan. Assessment of these risks will be continuous, and any issues that surface will be highlighted in the PMR presentations.

3.5 Performance Measurements

The success of the technology transition activities will be quantitatively assessed whenever practical. The following metrics will be tracked and reported periodically. Details of the measurement tracking, reporting, and instrumentation are contained in the Measurement Plan.

a. Number of media exposures for the SDRR Technology (papers, workshops, demonstrations, etc.).

b. Number of sites where SDRR is in-use.

c. Number of commitments in place (affiliates, strategic partnerships).

d. Qualitative and quantitative evaluation of SDRR technology products packaged for transition and marketing as documentation (technical reports, manuals, etc.), software, and training materials.
Qualitative evaluation will be based on the subjective evaluation criteria: integrated, complete, and usable.
Quantitative evaluation will be based on the number of products available for transition and marketing of the technology.

3.6 Deliverables

The major deliverables of the technology transition effort are the plans identified in Paragraph 3.2.1, and the Training Materials identified in Paragraph 3.2.4.

3.7 Resources Required

The Management Plan contains resource requirements (personnel, M&S, Travel, Tools, etc.) for this effort.

3.8 Interface Control

In addition to the interface control issues defined in the Management Plan, the technology transition activities require OGI to maintain an interface with existing technology transition agents, as defined in Paragraph 3.2.2, and with their affiliates and strategic partners.

OGI will develop the relationships with the technology transition agents, affiliates, and strategic partners during implementation of the technology transition activities. When these relationships are formalized, the interfaces between OGI and these outside organizations will be defined and documented.

between the saved file image and the most recently archived image of the same file. The extracted RCS log provides the numbers of lines added and deleted in each file image, relative to its immediate predecessor. This gives a finely detailed, time-stamped summary of editing activity. The environment only allowed the subject to have one file open at a time with write privileges, to make this kind of summary possible.

5.1 Method of analysis

Data that directly compared the performance of subjects over a series of trials was compared using an analysis of variance. This is a statistical test of the hypothesis that observed differences in the mean value of a performance metric cannot be accounted for by random variation in the observed values. In an analysis of variance it is important to eliminate all conceivable sources of systematic variation of the observables other than those that are being tested. Then the unaccountable variations that are observed can reasonably be assumed to be random, and normally distributed. Analysis of variance can take into account a single factor or multiple factors whose relationship to the observable is being tested. In the design of this experiment, the factors that could be correlated with observed differences in performance were the two technologies used and the individual differences among subjects. Other factors, such as order of use of the technologies, order of training in the technologies and prior familiarity with the particular task were eliminated in the design of the experiment. An unaccountable factor that contributed variability to the outcomes was the relative complexity of the individual design tasks.

6 Results of the experiment

6.1 Flexibility to meet varied specifications

Of the 12 original design tasks and 56 modifications, all were completed with the MTV Generator. However, MTV-G failed to generate Ada code for three of the original task specifications or for the modifications that derived from these. Acceptance testing was performed at the prototyping level for all tasks, and was repeated on the generated Ada code in all tasks for which code could be generated. No errors were found in the generated code.

While the experiment was in progress, some improvements to the SDRR translation pipeline that allowed an new version of MTV-G to generate Ada for all tasks that were assigned in the experiment. This new version of MTV-G was not made available to the subjects, however.

For the three original design tasks for which Ada code generation had failed in the experiment, the Ada code generated at OGI by the new version of MTV-G encountered capacity limits of the Ada compiler. We attribute this problem both to immaturity of the generator technology and to the fact that the Sun Ada compiler was not designed with generated code in mind, and imposes several seemingly arbitrary capacity limits.

The same design tasks were also performed with the MTV Model Solution. However, one of the original design tasks and the series of modification tasks that derived from it could not be completed because of an Ada compiler error for which no work-around was found.

Solutions were not found for two Model Solution modification tasks that required an out-of-scope feature to be implemented. This is not to say that it was impossible to implement this feature using the Model Solution, but merely that it was sufficiently difficult that no solution was found within the time allotted to the experiment.

In subsequent results that compare productivity and reliability measures of the two technologies, data from the design tasks that were not able to be completed in the Model Solution were eliminated in both technologies. This preserved the matching of factors on which the analysis depends, to the greatest possible degree.

4.1.3 Outline

Three aspects of technology transition are covered in this plan:

1. Maturation of the current SDRR tool suite so that the existing technology may be productized (Section 4.2);

2. Marketing and outreach so that the technology does not lie unused but finds industrial application (Section 4.3); and

3. Development of the SDRR method, so that the technology can be applied in a wider variety of areas and situations (Section 4.4).

4.2 MATURATION OF THE CURRENT SDRR TOOLS

The current SDRR tool suite is moderately stable and robust, but there are a number of improvements that need to be performed.

4.2.1 Goal

To license the current SDRR technology, and see it used in industry.

4.2.2 Plan

A plan for (a) maturing and (b) documenting the current tool suite to the point of licensibility will be developed and implemented. We will pursue our contacts with Open Systems Engineering (D.Phaneuf) to obtain entree to projects for commercial vendors.

4.3 MARKETING AND OUTREACH

4.3.1 Goal

To increase diversification of customer base by identifying and recruiting potential clients, collaborators, center advisors and center members from both industry and government.

4.3.2 Criteria for Success

1. Number of : PacSoft memberships, Contacts/Advocates (government and industrial), Clients, Collaborators and joint projects, and Center advisory board members

2. Technology marketing agreements stimulating rapid transition of new technology into practice.

4.3.3 Objectives

1 Year

- Develop two new government/industry contacts with the potential for small SDRR research contracts
- Form an advisory board of three industry advocates.

2 Years

- Develop one new government/industry advocate with the potential for a significant SDRR research contract
- Cultivate relationships (two contacts) with non-local industry to generate advocates in companies with potential interest in research
- Have an advisory board containing at least four advocates

3 Years

- Cultivate additional government/industry contacts, advocates and clients with focus on high-potential, large research contracts
- Have an advisory board containing at least six advocates

4.3.4 Approach

The most promising avenue for technology transition is the cultivation of personal relationships with potential clients who have both incentive to try our new technology to solve existing problems. Identification and recruitment of these contacts will entail comprehensive market research, a planned strategic program to increase visibility and presence in the industrial community, and deliberate effort to focus the scope of government/industrial contact effort to those with high potential. Recognizing that the development of a personal relationship requires much time, all PacSoft personnel will be expected to contribute to the marketing effort and to use marketing techniques to move beyond haphazard contact to a more focused and deliberate effort.

4.3.5 Work Program

4.3.5.1 Possible Insertion Points

Identification of characteristics of (i) problem domains amenable for SDRR development to be used in identifying potential clients, and (ii) organizations able to accept the new technology. This will lead to a document detailing criteria for evaluating suitability of problem domain for application of SDRR technology. Criteria should be as specific as possible, including minimum estimated labor hours for development and evolution, specific aspects of the problem domain that make it amenable to generator technology, and characteristics of an organization that might be able to make the evolutionary shift to use SDRR.

4.3.5.2 *Presentation Materials*
Design and produce a reusable marketing presentation and training for all faculty and staff designed to identify specific problems experienced by potential industry collaborators/clients and demonstrating how SDRR can be utilized to solve problems.

4.3.5.3 *Marketing Presentations*
Deliver a series of at least 12 market presentations during the first year at sites with potential research dollars and possible problem domains. Emphasis on listening to the customer rather than "selling" technology.

4.3.5.4 *Marketing Consultant*
Hire a marketing consultant to (a) provide support for training of PacSoft personnel in market research and developing contacts, (b) assist in the development of a comprehensive market plan, and (c) develop marketing literature. The literature describing PacSoft and the SDRR technology is broadly interpreted to include all communication sources such as brochures, World Wide Web (WWW), conference flyers, etc.

4.3.5.5 *Marketing Plan*
Develop a comprehensive marketing plan detailing specific 3-year goals and the methods by which they may be obtained.

4.3.5.6 *Identify Specific Target Markets*
Identify target markets in order to be able to focus marketing efforts. This should be an extension of the marketing plan and should be based on information gained during market presentations made previously.

4.3.5.7 *Advisory Board Plan and Charter*
Develop a charter for a PacSoft Advisory Board, describing relationship to PacSoft and long-term goals of board, together with a plan for populating the board: what backgrounds, how many advisors etc. The aim is for the board to be a formal mechanism for soliciting industrial input, as well as keeping industry appraised of our work.

4.3.5.8 *Solution-based Consulting Firm*
Development of relationship with solution-based consulting firms to assist in reaching potential customers, and obtaining detailed feedback of areas in which the technology needs to develop.

4.4 DEVELOPMENT OF THE SDRR METHOD

In contrast to simply maturing the current tool suite, there are a number of development directions for SDRR that will make the technology more powerful and more widely relevant.

4.4.1 Goal

To obtain (a) technology licensing agreements with commercial vendors of software tools, (b) cooperative joint projects with other research organizations, and (c) effective collaborations sufficient to produce joint research papers and proposals, with individual researchers outside PacSoft.

4.4.2 Criteria for Success

1. Research collaborations stimulating others to work on research problems related to PacSoft projects.

2. Mutual research projects with strong partners. Joint research proposals submitted.

3. A commercial organization providing on-going support of PacSoft developed software tools.

4.4.3 Objectives

1 Year

- PacSoft will secure a collaborative research agreement with another software research organization.
- PacSoft will obtain an agreement with a commercial software consultant to assist in technology transition.

2 Years

- PacSoft will initiate a joint research project with researchers from another institution

3 Years

- PacSoft will have transitioned its technology to a commercial vendor of software development tools.

4.4.4 Work Plan

4.4.4.1 NIST and ATP

Familiarize ourselves with NIST and ARPA ATP programs. Seek industrial partners and take the lead in organizing joint project proposal efforts.

4.4.4.2 Write Alternative SDRR Implementation Back-End

Write new back-end for the SDRR pipeline to produce C, C++ or Modula. This will demonstrate that we can be flexible with respect to the environment we target, and makes the technology more immediately applicable in fresh areas.

4.4.4.3 *Fresh Application Areas*

Find new application niches that look promising, e.g. 4GL producers. Examine creative opportunities created by new technology in hardware or software.

4.4.4.4 *Visitors*

PacSoft will step up its program of inviting short-term visitors from other research institutions to develop personal relationships.

4.4.4.5 *Collaboration Opportunities*

Many ARPA program initiatives encourage teaming between research organizations and industrial firms. PacSoft will track program announcements, paying special attention to opportunities in which we may collaborate with others to furnish unique research capabilities.

PacSoft will also work with a consulting organization, such as Open Systems Engineering, to seek opportunities for application of its technology to solve problems in commercial environments.

Faculty will search for proposed or currently on-going projects at other institutions that address complementary problems, seeking opportunities for synergistic collaborations

4.4.4.6 *Continued Productization*

As its program generation systems mature, PacSoft will work with commercial software tool vendors to seek commercialization opportunities.